Design and Implementation of a Scalable, Secure, and Maintainable Public API for Data Access in a Massively Multiplayer Online Role-Playing Game

Master Thesis

by

Pauline Röhr
569735

at the

Hochschule für Technik und Wirtschaft (HTW) Berlin
Faculty 4: Computing, Communication and Business
International Media and Computing

in cooperation with

Sandbox Interactive GmbH

1st Supervisor: Prof. Dr. Gefei Zhang
2nd Supervisor: Dipl.-Inf. David Salz

Abstract

Public game APIs are a valuable resource for players and developers alike, enabling third-party development and data access. Albion Online lacks an official API, leading to a reliance on unofficial means of data acquisition with security concerns.

This thesis presents the design and implementation of a scalable, secure, and maintainable public API for Albion Online. For this, the system integrates a data processing pipeline for data transformation, an API gateway for authentication and routing, and a RESTful API for game data access. Industry standards and best practices in security, maintainability, and performance optimisation guide the development.

The solution is evaluated against functional and non-functional requirements, highlighting improvements over existing unofficial methods. While the provided API implementation establishes a strong foundation, full deployment to external users requires further testing.

This work provides insights for game developers on opening game data while maintaining control and performance.

Contents

1 Introduction **1**

1.1 Problem Statement 1
1.2 Research Objectives 2
1.3 Scope and Limitations 2
1.4 Thesis Structure 2

2 Background and Related Work **4**

2.1 Albion Online 4
2.1.1 Game Architecture and Infrastructure 6
2.1.2 Web Architecture and Infrastructure 7
2.2 Public APIs in Online Games 8
2.2.1 Purpose and Use Cases 9
2.2.2 Challenges in Public Game APIs 11
2.2.3 Implications for Albion Online's API 11
2.3 Existing Work on Scalable and Secure API Design 12
2.3.1 Comparison of Notable APIs 12
2.3.2 Literature 13

3 Fundamentals **15**

3.1 API Design 15
3.1.1 API Paradigms 16
3.1.2 REST API Design Principles 17
3.1.3 Versioning 18
3.1.4 Security 19
3.1.5 Documentation 23
3.1.6 Caching 24
3.1.7 Scalability 25
3.1.8 API Gateways 28
3.2 Data Processing 28
3.2.1 ETL Architecture and Components 29
3.2.2 Scalability and Performance 31

3.3 Employed Technologies 32
3.3.1 Java Frameworks 33
3.3.2 REST API Frameworks and Tools 33
3.3.3 Data Processing 33
3.3.4 Data Storage 34
3.3.5 Infrastructure and Deployment 36

4 Analysis of the Current System **37**
4.1 The Game Info Service 37
4.1.1 Included Features 37
4.1.2 Usage 38
4.1.3 Tech Stack and Architecture 41
4.1.4 Resource Usage and Performance Metrics 42
4.1.5 Maintainability 43
4.1.6 Backward Compatibility 45
4.1.7 Lessons Learned 45
4.2 Third-Party Solutions 46
4.3 Available Infrastructure and Tools 46

5 Requirements **48**
5.1 Functional Requirements 48
5.2 Non-Functional Requirements 50
5.3 Constraints 52
5.4 Assumptions 53
5.5 Prioritisation and Scope 53

6 Architecture and Design **54**
6.1 System Architecture 54
6.2 API and Endpoint Design 57
6.2.1 Security 57
6.2.2 Rate Limiting 59
6.2.3 Versioning 59
6.2.4 Endpoint Design 59
6.3 API Key Management 64
6.4 Technologies and Frameworks 66
6.5 Database Schema 67
6.6 Data Processing 69

7 Implementation **71**
7.1 Service Setup 71
7.2 Authentication Service 72

7.3 API Gateway . . . 72
7.3.1 Dynamic Route Registration . . . 73
7.3.2 Security . . . 75
7.3.3 Rate Limiting . . . 76
7.4 ETL Process . . . 76
7.4.1 Spring Batch Configuration . . . 77
7.4.2 Performance Improvements . . . 77
7.4.3 Latency Issues . . . 80
7.5 REST API . . . 82
7.6 API Key Management . . . 83
7.7 Deployment . . . 84

8 Software Testing and Monitoring **85**
8.1 Test Plan . . . 85
8.2 Developer Testing . . . 87
8.2.1 Manual Testing . . . 87
8.2.2 Unit and Integration Testing . . . 87
8.3 Monitoring and Logging . . . 88
8.4 Upcoming Testing . . . 89

9 Evaluation **90**
9.1 Functional Evaluation . . . 90
9.2 Non-Functional Evaluation . . . 91
9.3 Comparison to the Game Info Service . . . 92
9.4 Comparison to other APIs . . . 93
9.5 Limitations . . . 94

10 Conclusion **95**
10.1 Summary of Contributions . . . 95
10.2 Limitations and Challenges . . . 96
10.3 Future Work . . . 96
10.4 Final Remarks . . . 96

Sources **97**

Glossary **103**

A Appendix **i**
A.1 Sample Kill Response from the Game Info Service . . . i
A.2 SecurityConfig class in the API Gateway . . . iv
A.3 IpAndApiKeyResolver class in the API Gateway . . . vii
A.4 Database Schema for Kills . . . viii

A.5 Kill Mapper x
A.6 Spring Batch configuration for kill event processing xii
A.7 Kill Processing Batch Scheduler xiv
A.8 Public API Metrics Notebook in DataDog xvi
A.9 Full Source Code xviii
A.10 Tools Used xviii

List of Abbreviations

API Application Programming Interface

CDC Change Data Capture

CDN Content Delivery Network

CPU Central Processing Unit

CRUD Create, Read, Update, Delete

CS Tool Customer Support Tool

DDoS Distributed Denial of Service

DNS Domain Name System

DOS Denial of Service

DTO Data Transfer Object

ELK Elasticsearch, Logstash, Kibana

ETL Extract, Transform, Load

GDPR General Data Protection Regulation

GIS Game Info Service

HTTP Hypertext Transfer Protocol

HTTPS Hypertext Transfer Protocol Secure

IP Internet Protocol

JDBC Java Database Connectivity

JPA Jakarta Persistence API

JSON JavaScript Object Notation

JWT JSON Web Token

MMORPG Massively Multiplayer Online Roleplaying Game

MVC Model-View-Controller

MVP Minimum Viable Product

ORDBMS Object-Relational Database Management System

ORM Object-Relational Mapping

OWASP Open Web Application Security Project

POJO Plain Old Java Object

PvP Player versus Player

QA Quality Assurance

QoL Quality of Life

RAM Random-Access Memory

REST Representational State Transfer

RPC Remote Procedure Call

SOAP Simple Object Access Protocol

SSL Secure Sockets Layer

TCP Transmission Control Protocol

TLS Transport Layer Security

ToS Terms of Service

TTL Time To Live

UDP User Datagram Protocol

UI User Interface

URI Uniform Resource Identifier

UUID Universally Unique Identifier

VM Virtual Machine

XML Extensible Markup Language

YAML Yet Another Markup Language

List of Figures

2.1 Official Artwork - Crafting a Sword in Albion Online 5
2.2 Official Artwork - Guild Seasons 6
2.3 Albion Online architectural overview, updated diagram from [60] 7

3.1 General structure of an API documentation, translated from [69, p. 272] 23
3.2 3-tier architecture, translated from [69, p. 323] 27

4.1 Anonymised interface of the Killboard showing the "Top PvP Kill this month" on the Albion Online website 39
4.2 Interface to create a new Solo Build in the Character Builder on the Albion Online website 40
4.3 Current architecture surrounding the Game Info Service (GIS) 42
4.4 Amount of requests per minute against all the Game Info Service (GIS); Exported from Datadog 43
4.5 Average response times (in seconds) of the Game Info Service (GIS) in the American environment; Exported from Datadog 43

6.1 Initial architecture plan for the API systems 54
6.2 Architecture Diagram 56
6.3 Mockup 1 of the API key management page 64
6.4 Mockup 2 of the API key management page 65
6.5 Mockup 3 of the API key management page 65
6.6 Mockup 4 of the API key management page 66
6.7 Schema in the `albion_stats` database, exported from JetBrains DataGrip 68
6.8 Schema of the `events` table in the `dataexport` database, exported from JetBrains DataGrip 69

7.1 Schema of an API Key, exported from JetBrains DataGrip 72
7.2 JProfiler Hot Spots in JDBC Connections in the Stats-ETL-Service 80
7.3 Kill Processing in all three live environments; Exported from Datadog . 81
7.4 API Key management interface on the website 84

List of Tables

2.1 Comparison of Official Game APIs [1, 9, 16, 31, 68] 9
2.2 Comparison of notable APIs [83, 80, 29, 27, 79, 51, 78] 13

3.1 Open Web Application Security Project (OWASP) Top 10 vulnerabilities in web applications and APIs [47, 48] 23
3.2 Comparing reverse proxies, load balancers, and API gateways [30] . . . 28

4.1 Third-Party consumers of the Game Info Service API and their features 41

5.1 Functional Requirements of the Public API, Part 1 49
5.2 Functional Requirements of the Public API, Part 2 50
5.3 Non-Functional Requirements of the Public API, Part 1 50
5.4 Non-Functional Requirements of the Public API, Part 2 51
5.5 Non-Functional Requirements of the Public API, Part 3 52
5.6 Constraints of the Public API 52

6.1 Kills endpoints in the Game Info Service, as documented by [24] 60

7.1 Throughput results of different configurations in the Stats-ETL-Service 79

8.1 Manual Test Plan for the Public API, Part 1 85
8.2 Manual Test Plan for the Public API, Part 2 86
8.3 Automated Tests in the relevant Java services 88

Listings

6.1 Example Request to Retrieve Kills 62
6.2 Example Response for Kills List 62
6.3 Example Reduced Player Data 62
6.4 Example Request for a Single Kill 63
6.5 Example Response for a Single Kill 63
6.6 Example Player Data 64

7.1 RegisterAtGateway annotation interface 73
7.2 Pseudocode RegisterRoutesOnStartup class 74
7.3 Configuration of the API Gateway client 74
7.4 Kill transformation method in the KillMappingService class 77
7.5 SQL Migration to alter sequence increments 78
7.6 Kill identifier field 78
7.7 pg_dump and pg_restore commands 82
7.8 KillController class in the Stats-Read-Service 82

8.1 SQL query to check for duplicates in the kills table 87

A.1 IpAndApiKeyResolver.java; Mostly written by Jakob Erdmann and Ivan Borgardt vii
A.2 KillMapper.java x
A.3 KillEventBatchConfiguration.java xii
A.4 KillBatchScheduler.java xiv

1 Introduction

The gaming industry has evolved rapidly, with Massively Multiplayer Online Roleplaying Games (MMORPGs) like Albion Online engaging millions of players worldwide [39]. These games rely heavily on player interaction and community participation, which are critical to their success [17]. Albion Online, developed by Sandbox Interactive GmbH, lacks a scalable, secure, and maintainable public Application Programming Interface (API) for accessing in-game data. This absence is notable given that public APIs can be considered an industry standard [1, 9, 16] to improve player engagement and foster community-driven content creation.

The absence of a public API limits Albion Onlines' players' ability to analyse performance, share game-related data, and develop third-party tools that expand the gaming experience. To overcome these limitations, this thesis will design and implement an official API that will provide access to in-game data such as player statistics, combat data, and economic transactions.

1.1 Problem Statement

The absence of a structured public API in Albion Online has led to several challenges:

- **Lack of Official Data Access:** Players and third-party developers lack a reliable method for retrieving in-game data, forcing them to use unofficial and unreliable means.
- **Security and Fairness Concerns:** Players resort to network scraping or botting due to the lack of official means of data acquisition. These activities are deemed security risks and provide unfair advantages in the game to players with advanced technical knowledge.
- **Scalability and Maintainability Issues:** An existing system provides access to in-game data but was not designed for external access, making it unsuitable for large-scale third-party use. The system is prone to breakage.
- **Developer and Community Limitations:** Without a structured API, third-party developers struggle to build the tools they desire, limiting community engagement and innovation.

The goal of this thesis is to design and implement a public API that addresses these challenges by providing secure and scalable access to Albion Online game data.

1.2 Research Objectives

To address the identified problems, this thesis aims to design a public API that adheres to industry standards and best practices while ensuring scalability, security, and maintainability. For this, authentication, rate-limiting and versioning mechanisms are applied. A data processing pipeline has been implemented to prepare game statistics and data for external consumption.

This thesis then evaluates the API's performance, reliability, and compliance with functional and non-functional requirements. The implemented solution is compared to existing APIs in the gaming industry and Albion Onlines' current unofficial game data access solutions.

1.3 Scope and Limitations

The primary focus of this thesis is the development of an Minimum Viable Product (MVP) API for data access, with data on in-game combat as the initial use case. While the API architecture is designed to support additional datasets, such as marketplace transactions and player statistics, these extensions are outside the scope of this thesis.

The project does not include full-scale penetration testing or production deployment of the API. While security measures have been implemented, further security audits and testing will be required before the API can be officially released.

1.4 Thesis Structure

This thesis is structured as follows:

- **Chapter 2** provides an overview of Albion Online, existing public APIs in online games, and related work on scalable and secure API design.
- **Chapter 3** introduces fundamental concepts, including API design principles, security considerations, and data processing pipelines.
- **Chapter 4** analyses the existing system, identifying its limitations and motivations for the new API.
- **Chapter 5** defines the functional and non-functional requirements for the public API.
- **Chapter 6** presents the design of the new API, including system architecture, security, and endpoint specifications.
- **Chapter 7** details the implementation of the API, including data processing, authentication, and performance improvements.

- **Chapter 8** describes the testing and monitoring approaches applied to ensure software quality and reliability.
- **Chapter 9** evaluates the API against its requirements, compares it to existing solutions, and discusses limitations.
- **Chapter 10** summarises the findings of this thesis, discusses its contributions, and outlines future work.

Structuring the thesis in this manner follows a logical progression from problem identification to solution development and evaluation. The findings presented here aim to serve as a foundation for future improvements and expansions of the Albion Online Public API.

2 Background and Related Work

In order to develop a scalable, secure and maintainable solution, it is necessary to understand the context of public APIs in massively multiplayer online role-playing games (MMORPGs). This chapter provides an overview of Albion Online, the game for which the proposed API is designed, explores the role of public APIs in gaming, and examines existing work on scalable and secure API design.

2.1 Albion Online

Albion Online is an MMORPG developed by Berlin-based studio Sandbox Interactive. Set in a medieval fantasy world, it is a sandbox game that affords players a great degree of freedom and creativity to engage with the game and each other as they see fit. In this way, the shape of the game world is primarily determined by the players themselves and their decisions, and it evolves dynamically.

According to the Cambridge Dictionary, an MMORPG is "a computer game that can be played by many people at the same time and in which players control the actions of characters in an imaginary world" [14]. MMORPGs typically feature character progression, in-game economies, and cooperative as well as competitive multiplayer gameplay. These features can be observed in genre representatives, for example World of Warcraft, Guild Wars 2, or Final Fantasy XIV.

Additionally, Albion Online is classified as a sandbox game, which is "a computer game, that allows you to play in any way you want, rather than having to follow set rules" [15]. Albion Online embodies this approach through its player-driven economy, full-loot Player versus Player (PvP) mechanics, and territory control systems, allowing players to forge the world of Albion and their gameplay to their desire.

Albion Online was first launched to the public in July 2017 [64], and since then, it has passed several significant milestones. In April 2019, the game went free-to-play, expanding its community to include new players worldwide [66], and in June 2021, the game was released for mobile devices. This allows players to play on both desktop and mobile, with the same characters and in the same game world, which marks Albion Online as the first true cross-platform MMORPG [67].

Playable in fifteen languages and with a global audience, in May 2024, Albion

Online surpassed 350,000 individual users daily – making it one of the world's biggest MMORPGs and one of Germany's most successful gaming exports [65, 43].

Being a sandbox MMORPG, the vast amount of experiences are made together with other players, whether it is a duel between two fighters, an ambush on a player transporting valuable goods, or buying and selling on the marketplace. At the heart of Albion lies its player-driven economy. Any armour, weapon, or food has been crafted by players (as shown in Figure 2.1). Players have sourced the ingredients for those items. Local marketplaces allow players to buy and sell items to each other. The players set the prices themselves, leading to a supply-and-demand economy with plenty of profit opportunities.

Figure 2.1: *Official Artwork - Crafting a Sword in Albion Online*

Another core feature of Albion Online is the territories. These are fortified outposts that guilds of several hundred players can feud over (Figure 2.2 shows promotional artwork for this feature). Holding high-quality territories rewards the guilds with points for seasonal competitions.

Figure 2.2: *Official Artwork - Guild Seasons*

Additionally, players can team up to hunt rare creatures for their ingredients, delve into dungeons to retrieve artefacts or explore an unstable network of tunnels built by civilizations long gone.

2.1.1 Game Architecture and Infrastructure

The development and operation of Albion Online involve a technical infrastructure designed to support its real-time multiplayer online features. Sandbox Interactive CTO David Salz has documented and justified the technical infrastructure on his blog[1]. The following paragraphs summarise his blog posts and talks and enrich the information to represent the system's current state. Many of the included technical decisions were made in 2012 when Sandbox Interactive first began developing the game.

Firstly, Unity3D was selected for its flexibility and cost-effectiveness. At the time, the engine already provided cross-platform support, which was not a planned feature but a promising opportunity for Albion Online [61, 60].

While Unity is a suitable choice for the game client to be run on users' devices, it is not "flexible, scalable and stable enough to support a game server for thousands of concurrent players" [60]. Instead, the game servers are written in C# and partly backed by the server framework available through Photon. The proprietary game server code includes collision detection, pathfinding, and level-loading implementations [61, 60].

Photon was selected as network middleware. Photon's reliable User Datagram Protocol (UDP) protocol is employed for most game communications, with movement

[1] davidsalz.de

messages sent as unreliable UDP to optimise performance. Transmission Control Protocol (TCP) is used for chat functionalities and inter-server communication [60].

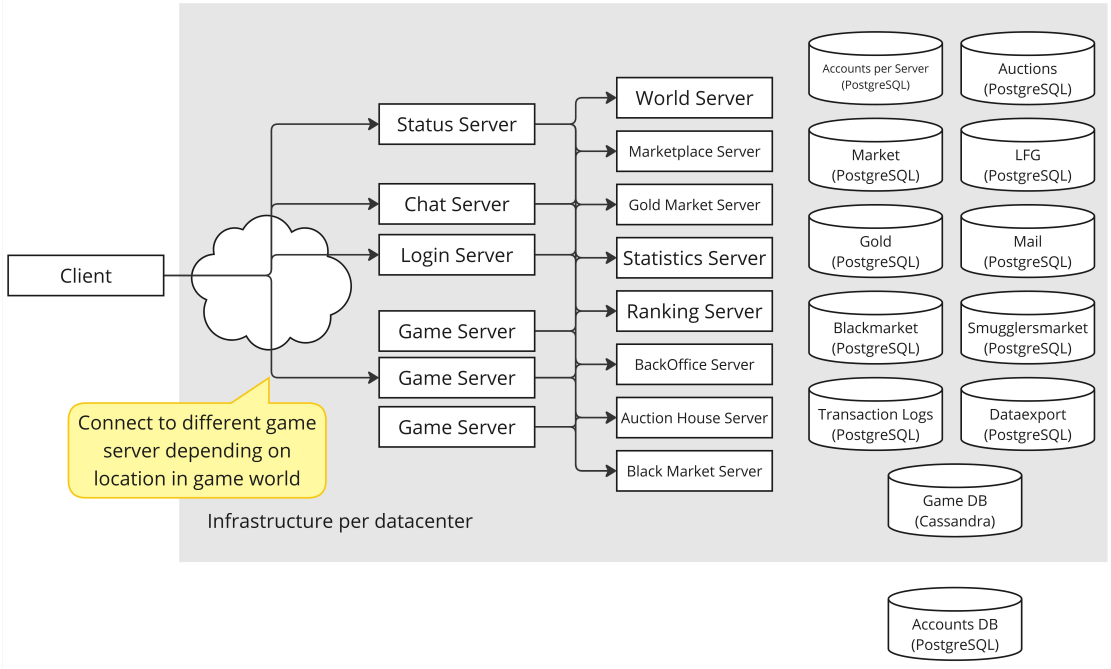


Figure 2.3: *Albion Online architectural overview, updated diagram from [60]*

Additionally, Apache Cassandra and PostgreSQL were selected for data storage. Apache Cassandra is a NoSQL database whose distributed architecture supports high throughput and horizontal scaling, aligning with the game's requirement for a potentially infinite world size. While Cassandra allows quick write processes and, consequently, high throughput, it cannot perform complex queries. Therefore, additional PostgreSQL databases are utilised for query-intensive operations, such as in-game marketplaces [60].

Albion Online's game servers are currently hosted on bare metal machines by GCore to meet the demands of strong CPU performance and memory requirements. The infrastructure is designed to distribute the game world across several physical servers [60]. Until 2023, Albion Online operated on a single global game server cluster in a data center in the US. In 2023 and 2024, server clusters in Singapore and the Netherlands have been introduced to allow players to play at lower latency globally [62, 63].

A rough architectural overview is depicted in Figure 2.3.

2.1.2 Web Architecture and Infrastructure

At Sandbox Interactive, game and web development are historically separated. When the game was first in development, all web topics were outsourced. The contracted company developed a PHP-based monolithic website, a Java- and Scala-based Cus-

tomer Support Tool (CS Tool), and set up off-the-shelf forum and wiki solutions. The outsourcing approach was expensive and led to difficulties when integrating web and game infrastructure with each other [61]. Over time, a web development department was formed at Sandbox Interactive, which has since maintained and improved upon the previously mentioned systems and implemented new features outside the game. Today, the web infrastructure includes features like a shop and payment system, mail and newsletter management, marketing tracking and a Twitch integration.

The initial setup of the web infrastructure followed a monolithic architecture approach, with both the PHP-based website and the CS Tool having access to all required data sources, handling backend and frontend code within the same project, and spanning many features. Especially for the website, it became clear early on that the included features were too many. Therefore, the web team started working towards a microservices architecture by splitting features off. Initially, this was done with the Dropwizard framework and later with Spring Boot 2 (see subsection 3.3.1).

Currently, the web infrastructure stands at the point of a "hidden" or a "modular" monolith [5]. Many of the services depend on each other; the website and CS Tool still utilise many services and partly have shared persistence.

Since a public API will not be integrated into the game directly, developing it falls into the responsibility of the web team at Sandbox Interactive. The current state of the system and previous work towards this goal are analysed more in-depth in chapter 4.

2.2 Public APIs in Online Games

Many of Albion Online's direct competitors and other online games already provide an official public API for their users (see Table 2.1). Those who do not receive requests for it from their respective communities[2] or have inofficial solutions implemented by their users in place[3]. The Albion Online community frequently asks for an official API as well[4], and has developed inofficial solutions that are further analysed in section 4.2.

These APIs provide access to various game-related functionalities, such as retrieving player statistics, accessing in-game marketplaces, or providing intel for PvP combat. This section explores the purpose, common use cases, and challenges associated with public game APIs. Due to the lack of scientific or academic work on MMORPGs or the relevance of public APIs for online games, most of the following claims are based on assumptions and experience gathered by the developers at Sandbox Interactive. This includes insights from Timothy Suthers, who has worked on the official Eve Online API

[2]Star Wars: The Old Republic: Forum Post, New World: Reddit Post, Final Fantasy XIV: Reddit Post, Elder Scrolls Online: Forum Post

[3]Star Wars: The Old Republic: https://ixparse.com/, Final Fantasy XIV: https://v2.xivapi.com/, Warframe: https://docs.warframestat.us/

[4]Reddit Post, Forum Post 1, Forum Post 2

in the past and has started working at Sandbox Interactive as a Senior Web Developer in 2025.

Game	Authentication	Data Scope	Rate Limits
EVE Online	OAuth 2.0	Player Statistics, Static Game Data, Market Data, PvP Data	Error Rate Limiting, Rate Limits on specific endpoints
Guild Wars 2	Scoped API Keys	Account and Player Data, Static Game Data, Market Data, PvP Data	300 requests per minute
World of Warcraft	OAuth 2.0	Account and Player Data, Static Game Data, Market Data	36,000 requests per hour, 100 requests per second
Lost Ark	OAuth 2.0	Player Statistics, Static Game Data, Market Data	100 requests per minute
Path of Exile	OAuth 2.0	Account and Player Data, Static Game Data, PvP Data	Dynamic Rate Limits

Table 2.1: *Comparison of Official Game APIs [1, 9, 16, 31, 68]*

2.2.1 Purpose and Use Cases

Depending on the game, public APIs serve various purposes, both for the game developers and the players. They encourage community engagement by offering access to game data, allowing players to interact with the game outside of the user interface. Third-party tools, wikis, and analytics platforms enable players to share knowledge, cooperate on strategy, and contribute to the game ecology. These interactions increase player retention and the game's long-term appeal [20].

From a development perspective, exposing game data through a public API allows studios to leverage third-party innovation, distributing the workload of tool creation and maintenance. Rather than dedicating internal resources to create and maintain every feature that players request, developers can enable community-driven solutions that expand the game [69, p. 10]. Complex tools, such as market activity trackers or crafting planners, are often built by community members who tailor their implementations to their own and to other players' needs (see, for example, subsection 4.1.2 and section 4.2). By facilitating external development like this, game studios can focus on core game improvements and expansions, while players still receive the specialised tools they desire.

Security concerns are another reason for offering an official public API. Without an official data source, players seeking game data may resort to botting, network scraping, or unauthorised packet interception practices [77]. By providing a secured and well-documented API, game developers can guide players toward legitimate means of data access. Insecure means of game data acquisition could, therefore, be prohibited through the game's Terms of Service (ToS), and any users still acquiring data through illicit methods can consequently be banned from the game.

Public APIs also enable a simple and cheap product research method. With third-party tools developed by the community, game developers can analyse included features and the interest each of them attracts. Any feature that gains traction in the community might be a candidate to be implemented directly into the game.

For players, public APIs introduce a range of applications that extend their gaming experience. One of the use cases for games that include PvP content is the collection and analysis of intelligence for combat. Access to killboards, combat logs, and player statistics allows competitive players to refine their strategies, analyse their performance, and anticipate enemy movements. This caters especially to the "Killer" archetype, as defined in Bartle's player taxonomy, where players seek to dominate others through skilful engagement and superior preparation [6].

Beyond combat, APIs enable players to optimise in-game efficiency, particularly in economic and crafting activities[5]. Tools that track auction house prices, suggest profitable crafting routes, or predict market trends allow traders and crafters to make informed decisions. This analytical approach resonates with "Achievers"-players motivated by progress, mastery, and in-game success—who rely on precise data to maximise their efficiency [6].

Public APIs also appeal to players who enjoy exploring game mechanics through data analysis. Whether tracking personal progress, visualizing item trends, or comparing historical performance, statistics-driven players benefit from structured access to game data. Additionally, third-party applications built on APIs can extend a game's reach beyond its native platform. Companion apps, browser-based tools, and integrations with external services allow players to interact with the game from anywhere, reinforcing engagement outside of active gameplay sessions[6].

In summary, public APIs bridge the gap between game developers and the player community. They empower players with tools to enhance their experience while alleviating development burdens and reducing reliance on unauthorised data extraction methods.

[5] see for example gw2efficiency.com

[6] Warframes' companion app is an example of this

2.2.2 Challenges in Public Game APIs

While official game APIs offer multiple advantages for game developers and players alike, their implementation also entails challenges. Firstly, a public API can introduce new vulnerabilities. It could act as an interface for Distributed Denial of Service (DDoS) attacks (see chapter 4), which is why the corresponding architecture should be set up to protect internal services and the game servers from possible impact. Additionally, any API that provides player or account data should utilise established security practices and authorization to protect user privacy.

Other than the risk of technical abuse, players might be able to use the provided data from an API in unintended ways. Unrestricted access to all game data can have implications on game balance and player behaviour, which means that the provided endpoints in an API should be thoroughly selected and planned beforehand. Changes in player behaviour were, for example, noticed in the game Eve Online:

> Eve PvP has largely degenerated into ambushes and counter-ambushes, as people use all the data on hand to avoid fights they aren't sure they can win.
>
> One of the largest fights in the game happened when both sides made the calculation incorrectly and got stuck in a fair fight, the worst kind of fight to be stuck in.
>
> (Timothy Suthers, personal communication, March 2025)

Therefore, the applied game design should align with the overall game vision and prevent interactions that act as a detriment to player enjoyment.

Another challenge for public game APIs is the additional operational and technical cost. The design, development, maintenance and support of an API must be incorporated into the game studio's production planning and budget. This also includes costs incurred for computing power, bandwidth or data storage.

Despite these challenges, public APIs can remain a valuable tool for game developers willing to invest in their proper management. By proactively addressing security, infrastructure demands, versioning, and game balance concerns, developers can create an API that benefits both the player community and the game's long-term success.

2.2.3 Implications for Albion Online's API

Having analysed the advantages and disadvantages of a public API in online games, the implications for the API design and development for Albion Online can be deduced.

Public API in online games cater towards the "Killer" and "Achiever" archetypes [6], who are both very prominent in Albions' target audience[7]. These player groups can be

[7]This was confirmed by a game designer at Sandbox Interactive

activated and engaged more by providing official means to access game data.

Additionally, once third-party developers for Albion Online receive access to an intentional and secure API, they will likely move away from illicit means such as packet interception or botting. As a result, the game programmers at Sandbox Interactive would be able to secure the in-game chat and the network traffic more heavily, increasing the general security of the game in turn. In the past, encryption was added to some game data as a measure against botting. This resulted in many third-party tools breaking (see section 4.2). The encryption was then removed again until a different solution to access the data could be provided to third-party developers. Therefore, a public API also reduces the risk of compatibility issues when game changes occur.

The Albion Online designers might be able to leverage another advantage of an API through tracking and analysing tools developed by third parties. This analysis can be used to identify features that are of interest to the community, whether those are Quality of Life (QoL) improvements or new game mechanics.

Lastly, the design of a public API for Albion Online should carefully consider which data to expose and implement security techniques to prevent abuse by malicious players.

2.3 Existing Work on Scalable and Secure API Design

Apart from research within the games industry, the design and development of an API can be supported by more general resources from the software engineering field. This section explores a selection of highly used APIs by other companies and literature on the topic of scalable and secure API design.

2.3.1 Comparison of Notable APIs

By taking a look at established APIs, common and best practices can be deduced. Some of the APIs included in Table 2.2 serve as the main product of their respective companies and can, therefore, be used to model the design and development of an API [69, p. 11]. As a basis, the authentication methods, rate limits and included use cases are examined, since these points provide the most insight into the properties of a scalable and secure API (see section 3.1).

As can be seen in Table 2.2, API keys and OAuth 2.0 are the most prominent authentication methods. Both methods have their respective advantages and disadvantages, which will be further explained in subsection 3.1.4.

Another inference is the practice of rate limiting per use case. Many of the examined APIs apply rate limits depending on the complexity of underlying calculations. That way, endpoints that require more computing resources to return a response have lower rate limits. This technique is helpful for API scalability and will also be further discussed in subsection 3.1.4.

Lastly, the comparison shows that the most common use cases cover data retrieval functionality, whereas only a few of the selected APIs provide automation endpoints or Remote Procedure Calls (RPCs). This is likely due to the comparatively low complexity of data retrieval APIs.

API	**Authentication**	**Rate Limiting**	**Selection of Use Cases**
GitHub API	OAuth 2.0, API Key	5,000 requests/hour and dynamic secondary rate limits	Repository management, CI/CD automation
Google Maps API	API Key, OAuth 2.0	Pay-per-use, limits depend on sub-API	Maps, routes, place information, environmental data
SendGrid API	API Key	Varying limits per endpoint	Transactional emails, newsletter management, marketing automation
Steam Web API	API Key	100,000 requests/day	Game news feeds and stats, user information
PayPal API	OAuth 2.0	Dynamic throttling	Payment processing and tracking, invoicing, subscriptions
Twilio API	API Key	Pay-per-use and varying limits per use case	SMS, voice, and video communication, marketing automation
Twitter (X) API	OAuth 1.0 & 2.0, Basic Authentication	Tiered limits	Social media data, analytics, user engagement

Table 2.2: *Comparison of notable APIs [83, 80, 29, 27, 79, 51, 78]*

2.3.2 Literature

In addition to modelling APIs after established implementations, developers can use a wide range of API design literature. O'Reilly Media, Manning Publications, Packt, and Apress are publishing houses known for their programming and informatics resources, and they all have published various books on APIs:

- J. Webber, S. Parastatidis, and I. Robinson. *REST in Practice: Hypermedia and Systems Architecture*. Theory in practice series. O'Reilly Media, 2010.

- M. Masse. *REST API Design Rulebook: Designing Consistent RESTful Web Service Interfaces*. O'Reilly Media, 2011.
- L. Richardson, M. Amundsen, and S. Ruby. *RESTful Web APIs: Services for a Changing World*. O'Reilly Media, 2013.
- K. Hunter. *Irresistible APIs: Designing web APIs that developers will love*. Manning, 2016.
- B. Jin, S. Sahni, and A. Shevat. *Designing Web APIs: Building APIs that developers love*. O'Reilly Media, 2018.
- N. Madden. *API Security in Action*. Manning, 2020.
- J. Gough, D. Bryant, and M. Auburn. *Mastering API Architecture: Design, Operate, and Evolve API-Based Systems*. O'Reilly Media, 2021.
- S. Sharma. *Modern API Development with Spring and Spring Boot: Design highly scalable and maintainable APIs with REST, gRPC, GraphQL, and the reactive paradigm*. Packt Publishing, 2021.
- B. De. *API Management: An Architect's Guide to Developing and Managing APIs for Your Organization*. Apress, 2023.
- B. Pedro. *Building an API Product: Design, implement, release, and maintain API products that meet user needs*. Packt Publishing, 2024.
- C. Staveley. *API Security for White Hat Hackers: Uncover offensive defence strategies and get up to speed with secure API implementation*. Packt Publishing, 2024.

The work on this thesis and the accompanying project are also guided by the book *API-Design: Praxishandbuch für Java- und Webservice-Entwickler* by Kai Spichale, published by dpunkt.verlag in 2019.

Additionally, companies like Google and Zalando have provided guidelines for API design and development. These can be useful resources to discover best practices:

- https://opensource.zalando.com/restful-api-guidelines/
- https://cloud.google.com/apis/design

3 Fundamentals

Developing a scalable, secure, and maintainable public API for an MMORPG requires an understanding of each of these characteristics. This chapter thus introduces the key concepts on API design to ensure said scalability, security and maintainability. Industry standards and best practices to develop an API and the required data processing pipeline are laid out. Furthermore, the chapter delves into the technologies that are integral to the project.

3.1 API Design

Modern software architectures rely on well-designed APIs to enable seamless communication between distributed systems. According to Bloch [10], two fundamental questions must be answered affirmatively to define an Application Programming Interface (API):

1. Does it provide a set of operations defined by their inputs and outputs?
2. Does it admit reimplementation without compromising its users? [10]

In the context of MMORPGs, public APIs expose game data to external applications, such as player statistics, marketplace information, and guild interactions (see section 2.2). To measure an API's effectiveness, any software quality model, like ISO/IEC 25010, can be taken into account. Spichale [69] identifies the following attributes as markers for the quality of APIs specifically:

- APIs must be complete and correct.
- APIs should be consistent, intuitively understandable, documented, minimal, stable, extensible and easy to learn. They should make it easy for users to write readable code. It should be difficult to use them incorrectly.
- APIs should be efficient and scalable.
- APIs should be reliable. [69]

A robust API design ensures adherence to these guidelines, enabling services to remain maintainable and extendable over time. This section explores the core principles of API design, focusing on aspects critical to developing a scalable, secure, and maintainable

public API for an MMORPG. It introduces relevant API paradigms and design principles, security considerations, performance optimisations, and documentation practices essential for facilitating developer adoption and long-term usability.

3.1.1 API Paradigms

The choice of an API paradigm fundamentally influences how software components interact and exchange data. APIs can be separated into two categories: programming language APIs and remote APIs. The latter can further be subdivided into REST, Simple Object Access Protocol (SOAP), messaging APIs, RPCs, and GraphQL-based APIs [36, 69]. While programming language APIs facilitate in-process communication within a single application, they are not relevant in this context, as the objective is to design a public API that enables structured data access over the web. All official game APIs investigated in section 2.2 fall into the category of REST APIs. Due to this, remote API paradigms besides REST will also not be further discussed.

REST

REST is an architectural style that "provides a set of architectural constraints that, when applied as a whole, emphasises scalability of component interactions, generality of interfaces, independent deployment of components, and intermediary components to reduce interaction latency, enforce security, and encapsulate legacy systems" [25, p. 105]. Due to these attributes, REST is widely adopted. The core principles of REST are:

- "Unique identification of resources
- linking/hypermedia
- usage of standard methods
- different representations of resources
- stateless communication" [76, p. 11, translated]

When this architectural style is applied to Hypertext Transfer Protocol (HTTP), it is called RESTful HTTP. Since HTTP is the most common basis for REST APIs, "REST" and "RESTful HTTP" shall be understood synonymously hereafter [26, 69, 76].

Given the need for an API that is scalable, lightweight, and widely compatible with existing web technologies, REST was selected as the preferred architectural style for this project. Its stateless nature supports high-concurrency MMORPG interactions, while standard HTTP methods enable efficient resource manipulation and caching.

Since REST principles can be implemented to varying degrees, the Richardson Maturity Model provides a framework to assess an API's compliance. Each level represents a step towards full RESTfulness, with increasing quality and usability benefits [58, p. 6].

- Level Zero: One Uniform Resource Identifier (URI), one HTTP method

- Level One: Many URIs, one HTTP method
- Level Two: Many URIs, each supporting multiple HTTP methods
- Level Three: Hypermedia; Resources describe their own capabilities and interconnections. [57]

While Level Three is a theoretical ideal, most real-world APIs remain at Level Two due to implementation complexity. Few APIs fully adopt hypermedia-driven navigation, despite its benefits in discoverability and client adaptability [58].

3.1.2 REST API Design Principles

In alignment with the core principles of REST, a set of best practices can be deduced and shall be considered when designing the public API for Albion Online.

Firstly, every resource should be uniquely addressable via a conventionally structured URI as defined in RFC3986, consisting "of a hierarchical sequence of components referred to as the scheme, authority, path, query, and fragment" [8]. In addition to this base structure, the following guidelines should be adhered to:

- Use lowercase letters.
- Use hyphens (-) instead of underscores (_).
- Avoid trailing forward slashes (`/`).
- Do not include file extensions; rely on content negotiation.
- Use a dedicated api subdomain (e.g., `api.example.com`). [41, p. 12f]

When defining an URI for an endpoint, a forward slash separator(`/`) should indicate a hierarchical relationship [41, p. 11]. Therefore, version numbers should not be embedded in the URI (e.g. `/v1/resource`, see subsection 3.1.3).

Resource URIs should never include CRUD terms, to ensure that the API remains resource-oriented rather than action-driven. Instead, HTTP methods should indicate which CRUD function is performed [41, p. 18]. This follows REST's core principle of a uniform interface, ensuring consistent interaction with resources. The following conventions have been established:

- "`POST` should be used to create new resources within a collection.
- `GET` should be used to retrieve a representation of a resource.
- `PUT` should be used to update or create resources.
- `DELETE` should be used to remove a resource from its parent.
- `OPTIONS` should be used to retrieve the available interactions of a resource.
- `HEAD` should be used to retrieve metadata of the current state of a resource" [58, p. 4].

REST APIs typically include *documents* (single instances of a resource), *collections* (groups of related resources), *stores* (client-managed resource repositories) and *controllers*

(for actions that do not fit standard HTTP methods). Following best practices, singular nouns should be used for individual resources and plural nouns for collections and stores, while verbs should be reserved for controller endpoints [41, p. 17].

In compliance with the REST principle of utilising standard methods, standardised HTTP status codes should be returned on all requests. The following list provides an overview of the most important status codes for this project:

- `200 OK` – The request was successful, and the server returned the expected response.
- `201 Created` – The request successfully created a new resource, and the response includes its location. This is commonly used for `POST` requests.
- `202 Accepted` – The request has been accepted for processing but has not yet been completed.
- `204 No Content` – The request was successful, but no response body was returned. This is useful for `PUT` and `DELETE` calls.
- `400 Bad Request` – The server could not process the request due to client-side errors, such as invalid parameters. The response should include further information on the cause of the error for the client.
- `401 Unauthorized` – Authentication is required or has failed. The client must provide valid credentials.
- `403 Forbidden` – The client is authenticated but lacks permission to access the requested resource.
- `404 Not Found` – The requested resource does not exist on the server.
- `410 Gone` – The requested resource has been removed and is no longer available. This can be used for a deprecated API endpoint.
- `429 Too Many Requests` – The client has exceeded the defined rate limit and should slow down its requests.
- `500 Internal Server Error` – An unexpected error occurred on the server.
- `503 Service Unavailable` – The server is temporarily unable to handle the request, often due to maintenance or overload.
- `504 Gateway Timeout` – The gateway or proxy server did not receive a timely response from an upstream service. [69, p. 196ff]

When designing and implementing a REST API, these industry standards and best practices should be followed.

3.1.3 Versioning

As APIs evolve, managing changes while ensuring backward compatibility is crucial to preventing disruptions for existing consumers. API versioning is a structured approach to introducing modifications without breaking client integrations. Given the long-term nature of public APIs, versioning enables iterative improvements while maintaining

service stability. Therefore APIs should be versioned from the start. Changes that require a new version include:

- Removal of a field in a resource
- Moving a field in a resource
- Renaming a field in a resource
- Changing the URI structure (in case no hypermedia is used)

On the other hand, adding a new resource or a new HTTP method is backwards compatible, and therefore does not require a version upgrade. [69, p. 200f]

The following version identification strategies are commonly used in REST APIs:

- **Query Parameter Versioning**: Including the version as a query parameter (e.g., `/resource?version=2`). While easy to implement, query parameters are intended for filtering, searching or sorting, not for version selection. Additionally, this technique may introduce caching issues.
- **Custom-Header-Based Versioning**: Specifying the API version in HTTP headers (e.g., `Accept: application/vnd.api+json; version=2`). This keeps URIs clean but requires clients to handle custom headers properly.
- **Content Negotiation**: Encoding the version within the `Accept` header (e.g., `application/com.albiononline.v2+json`). This method aligns well with REST principles but introduces complexity.
- **URI Versioning**: Inserting the version number directly in the endpoint path (e.g., `/v1/resource`). This approach is simple and widely adopted but does not strictly adhere to REST principles and is therefore semantically incorrect [69, p. 203ff].

In addition to the selection of a version identification strategy, APIs should implement deprecation policies to transition clients smoothly. Best practices include providing clear documentation on deprecated endpoints, setting `Deprecation` and `Sunset` HTTP headers, and collecting client consent on deprecation time spans [84]. This shows that API versioning is not strictly a technical topic, but also an aspect of the API owners relationship with their users. It is therefore recommended to provide a minimum time frame of endpoint validity when publishing an API or a new API version. Establishing a dedicated communication channel can reduce clients' risk of using outdated API versions longer than intended. [56].

3.1.4 Security

Security is a fundamental aspect of REST API design, particularly in the context of a public API for an MMORPG, where user accounts and transactional information must be protected against unauthorised access and sensitive in-game information should

be protected from exploitation. A well-secured API fosters trust among users and third-party developers. Common security principles and best practices are discussed in the following paragraphs, including Authentication, Authorisation, Encryption and Rate Limiting. Finally, the most common security vulnerabilities that should be avoided by API developers are listed.

Authentication is a mechanism to verify the identity of clients. Several approaches exist for securing APIs through authentication, each with distinct advantages and trade-offs.

- **HTTP Basic Authentication:** Name and password are Base64-encoded and transferred via `Authorization` header. This is very simple to work with, both for Client and Server. Since Base64 can be decrypted by anyone, Basic Authentication should only be used in combination with Secure Sockets Layer (SSL)/Transport Layer Security (TLS) to add a proper encryption layer [69]. This approach hinders third-party client development, since users should never hand their credentials to third parties. However, it can be useful in APIs for which each user writes their own client.
- **OAuth:** Defined in RFC 6749, OAuth 2.0 is a widely adopted authorisation framework that enables third-party applications to securely access API resources without exposing user credentials. OAuth 2.0 supports access tokens with various scopes, expiration policies, and four different authentication processes [18]. OAuth is of high complexity and produces a lot of overhead work for potential clients.
- **API Keys:** Simple, static keys used for authentication, often included in request headers. While easy to implement, they lack fine-grained access control out of the box and require third-party clients to have secure storage to keep users' keys safe.
- **JSON Web Token (JWT):** JWT is a compact and self-contained token format for securely transmitting authentication and authorisation information. JWTs can be signed and optionally encrypted, ensuring both integrity and confidentiality [42]. Authentication via JWT is a variant of API key authentication, where the JWTs are often transferred via the `Authorization` header as a Bearer token.

While authentication ensures that clients are who they claim to be when sending a request, authorisation examines whether clients have permission to execute interactions or read resources. According to Madden [40], two primary access control methods are utilised in APIs: "Identity-based access control first identifies the user and then determines what they can do based on who they are. A user can try to access any resource but may be denied access based on access control rules. Capability-based access control uses special tokens or keys known as capabilities to access an API. The capability itself says what operations the bearer can perform rather than who the user is. A capability both names a resource and describes the permissions on it, so a user is not able to access any resource that they do not have a capability for." [40, p. 22]

APIs should require authentication for all endpoints that handle sensitive or personal data, enforcing authorisation rules to restrict access based on user roles and permissions. When implementing any of these authentication approaches, developers should be aware of common mistakes causing vulnerabilities. According to the Open Web Application Security Project (OWASP) "An API is vulnerable if it: [...]

- Sends sensitive authentication details, such as auth tokens and passwords in the URL.
- Allows users to change their email address, current password, or do any other sensitive operations without asking for password confirmation.
- Doesn't validate the authenticity of tokens." [47]

Encryption

Aside from data protection within the API, security mechanisms should be applied to protect the data when travelling between the API and clients. In the case of REST APIs, encrypting the communication is a common practice. Without encryption, sensitive information such as authentication credentials and user data could be intercepted or altered by malicious actors, leading to data breaches and unauthorised access. To protect data in transit, REST APIs should enforce the use of Hypertext Transfer Protocol Secure (HTTPS) for all communications [40]. HTTPS is an implementation of the TLS protocol, which encrypts HTTP traffic. This enables secure communication between clients and servers, preventing eavesdropping and man-in-the-middle attacks.

Rate Limiting

Rate limiting is a mechanism that controls the number of requests a client can make to an API within a given time frame. It is crucial for maintaining API availability, preventing abuse, and ensuring fair resource distribution among clients. Without rate limiting, a single misconfigured or malicious client could overwhelm the API, leading to degraded performance or downtime. In the context of an MMORPG, where APIs may serve thousands of concurrent users retrieving marketplace data, tracking leaderboards, or accessing player statistics, rate limiting is vital in preserving service stability.

Rate limits can be applied at different levels, depending on API design and business requirements. A global rate limit applies uniformly across all endpoints, whereas a granular rate limit restricts specific endpoints based on their computational cost [38, p. 103]. For example, an API that provides game marketplace data might enforce stricter rate limits than a basic player profile retrieval endpoint. Rate limits can also be applied per user, per application, or per IP address, depending on authentication requirements. Authenticated APIs often use per-user or per-application rate limits, while unauthenticated APIs default to IP-based rate limiting. [38, p. 104]

Several algorithms exist for implementing rate limiting, each suited to different traffic patterns:

- **Token Bucket:** Clients receive a finite number of tokens that replenish over time. Requests consume tokens, and when the bucket is empty, requests are throttled. This approach allows short bursts of traffic while maintaining an overall limit.
- **Fixed Window:** Requests are counted within a fixed time window (e.g., 100 requests per minute). While simple to implement, this method allows burst traffic at the edges of the window.
- **Sliding Window:** Instead of resetting counts at fixed intervals, this approach maintains a rolling window, providing a more evenly distributed request rate. [38, p. 105ff]

When defining a rate-limiting policy, API designers should balance infrastructure protection with developer usability. A policy should be clear, predictable, and easy to work with, ensuring developers can anticipate limits and design their integrations accordingly. For usability, APIs should return appropriate HTTP status codes when rate limits are exceeded. The standard response is HTTP `429 (Too Many Requests)`, often accompanied by a `Retry-After` header indicating when clients may retry their requests [38, p. 110].

Finally, exceptions may be necessary for trusted partners or high-priority applications. Some APIs allow developers to request quota increases based on validated business needs. However, granting exceptions should be carefully considered to prevent unfair resource allocation and system overload.

By implementing a well-designed rate-limiting strategy, REST APIs can maintain responsiveness, fairness, and security while accommodating diverse client needs.

Common API Security Vulnerabilities

The OWASP conducted an analysis and collation of the Top 10 vulnerabilities in web applications and APIs (see Table 3.1). Developers should be aware of these vulnerabilities and employ appropriate measures to circumvent them. It is important to note that while every vulnerability in the Top 10 is worthy of attention, avoiding these alone will not guarantee the security of an application [40, p. 40].

By adhering to these security best practices, a REST API can minimise attack vectors, protect sensitive information, and ensure a secure and trustworthy interaction model for both clients and third-party developers.

Top 10 in Web Applications	Top 10 in APIs
A01:2021 - Broken Access Control	API1:2023 - Broken Object Level Authorization
A02:2021 - Cryptographic Failures	API2:2023 - Broken Authentication
A03:2021 - Injection	API3:2023 - Broken Object Property Level Authorization
A04:2021 - Insecure Design	API4:2023 - Unrestricted Resource Consumption
A05:2021 - Security Misconfiguration	API5:2023 - Broken Function Level Authorization
A06:2021 - Vulnerable and Outdated Components	API6:2023 - Unrestricted Access to Sensitive Business Flows
A07:2021 - Identification and Authentication Failures	API7:2023 - Server Side Request Forgery
A08:2021 - Software and Data Integrity Failures	API8:2023 - Security Misconfiguration
A09:2021 - Security Logging and Monitoring Failures	API9:2023 - Improper Inventory Management
A10:2021 - Server-Side Request Forgery	API10:2023 - Unsafe Consumption of APIs

Table 3.1: *OWASP Top 10 vulnerabilities in web applications and APIs [47, 48]*

3.1.5 Documentation

To design and develop an API for a public audience, documentation is required. Without knowledge of the available endpoints or how to reach them, users would be unable to request their data or develop third-party clients. Comprehensive and well-structured API documentation is therefore critical for adoption, usability, and long-term maintainability. Proper documentation reduces onboarding friction for developers, minimises support requests, and ensures consistency in API usage [69, p. 271].

Effective documentation should be tailored to the expected user group. The general structure of API documentation can be seen in Figure 3.1.

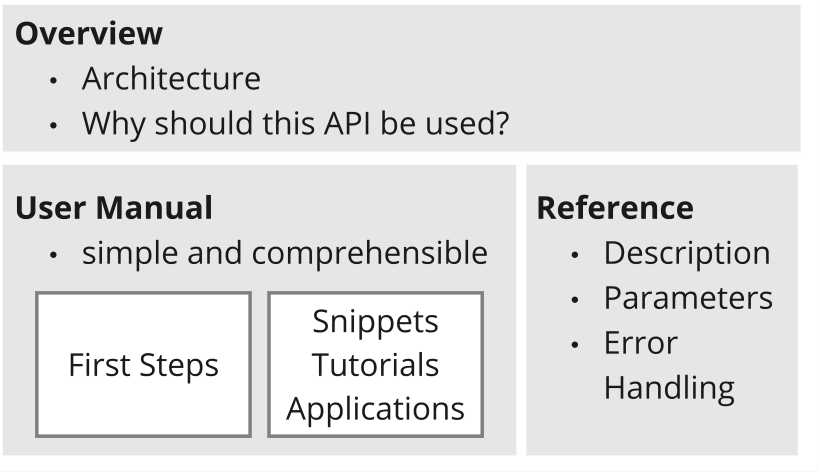


Figure 3.1: *General structure of an API documentation, translated from [69, p. 272]*

For REST APIs specifically, the documentation should cover authentication, use cases, paths, parameters, HTTP methods, request and response headers, status and error codes, payloads of requests and responses, and rate limits. In an effort for

standardisation, technologies have emerged that are commonly used for REST APIs documentation. The most common tools include:

- **OpenAPI (Swagger)**: A schema modeling framework, that allows documentation to be generated from Yet Another Markup Language (YAML) or JavaScript Object Notation (JSON) [32, p. 149f]. OpenAPI supports interactive documentation through tools like Swagger User Interface (UI), enabling users to explore endpoints and test requests without writing a client.
- **RAML (RESTful API Modeling Language)**: A human-readable format focused on designing and documenting APIs before implementation. The specifications are written in YAML and Markdown [32, p. 138f].
- **API Blueprint**: An open-source Markdown-based format. It can be integrated into the manufacturer platform (apiary.io) to generate server mocks and documentation [69, p. 292]
- **apiDoc**: Allows the user to document REST APIs directly in the source code. This is very similar to Javadocs, and supports all DocStyle capable languages, as well as Coffeescript, Elixir, Erlang, Perl, Python, Ruby and Lua. [69, 59]

Regardless of the chosen format, API documentation should be maintained as part of the development lifecycle, ensuring it remains accurate with evolving implementation.

3.1.6 Caching

Cache is a fast buffer memory, that can be used to store data from previous read operations and calculations. Caches are commonly used to improve performance and scalability in all types of software. For REST APIs, "caching response data can help to reduce client-perceived latency, increase the overall availability and reliability of an application, and control a web server's load" [41, p. 4]. Three cache topologies have emerged as part of the common web infrastructure: Local caches, proxy caches, and reverse proxy caches.

"A local cache stores representations from many origin servers on behalf of a single user agent, application, or machine. A consumer may have a local cache so that frequently accessed resources are stored locally and served immediately. Local caches can be held in memory or persisted to disk" [81, p. 160]. This can include private data, that would be inappropriate to store on the server side. Local caches are commonly integrated into web browsers [69, p. 298].

Data that is equal for all clients can be stored in a so-called proxy cache. "A proxy cache stores representations from many origin servers on behalf of many consumers. Proxies can be hosted both inside the corporate firewall and outside." [81, p. 160]

"A reverse proxy, or accelerator, stores representations from one origin server on behalf of many consumers. Reverse proxies are located in front of an application or web

server. Clusters of reverse proxies improve redundancy and prevent popular resources from becoming server hotspots" [81, p. 160]. Common implementations of (reverse) proxies are Varnish, Squid and nginx [69, 81].

Additionally, the usage of a Content Delivery Network (CDN) can help to deliver resources globally quickly. These networks consist of multiple servers in various geographic regions. When a client requests a resource, the closest server in the network is selected. [69, p. 303]

These topologies can be combined in any way, but developers should be aware that each layer of caching also adds a level of complexity when troubleshooting arising issues.

The following practices are recommended to realise a caching strategy:

- Usage of stable URIs, to increase cache hit rate
- Provide clients with validation tokens per resource so that the client only receives a new version of the requested resource if the token is invalid
- Identify resources that are equal for all clients
- Determine the Time To Live (TTL) for each cached resource, depending on how up-to-date the data needs to be
- When providing a REST API, setup a reverse proxy cache first. Extend this cache strategy as needed [69, p. 303f].

3.1.7 Scalability

As part of the design of an API, developers should consider the scalability of their system. "Scalability is the ability of a hardware and software system to increase its performance proportionally by adding resources within a defined scope." [69, p. 306, translated]. Need for scalability stems from the need for availability and reliability. If an API is suddenly used more than expected, a scalable system is able to adapt more easily. Since it is generally difficult to estimate the load that will be put onto an externally available system, the system should be designed in a scalable way. For this, various techniques can be applied.

An application can be scaled in two ways: vertically and horizontally. To scale vertically, a system component receives more resources, meaning more Central Processing Unit (CPU), Random-Access Memory (RAM) or disk storage. This is easy to do, but is quickly limited by hardware options or budget [69, p. 306f]. Additionally, the corresponding code needs to be able to make use of the given resources, for example through parallelisation on code level.

Horizontal scaling on the other hand is achieved by adding new nodes to the system. This requires the application to be parallelisable. For this, decentralised algorithms should be used. According to Tanenbaum and Steen [71], the following characteristics

are generally exhibited by these algorithms, distinguishing them from centralised algorithms:

1. No machine has complete information about the system state.
2. Machines make decisions based only on local information,
3. Failure of one machine does not ruin the algorithm.
4. There is no implicit assumption that a global clock exists.
5. Communication with and within the system needs to be stateless. [71, p. 11]

Horizontal scaling opens up the need to distribute requests between the existing nodes. This challenge is addressed through load balancing, which directs traffic to different servers based on predefined strategies. Various load-balancing techniques exist, each with distinct advantages and trade-offs depending on system architecture and traffic patterns. The most commonly used load-balancing approaches include:

- **DNS Load Balancing:** By registering multiple entries under a domain, the Domain Name System (DNS) can adopt the role of a load balancer. When a DNS lookup is done, a list of Internet Protocol (IP) addresses is returned, of which the order changes depending on varying parameters. This is very easy to implement but does not take server load into account, and it might take up to two days to apply any changes to the registered IP addresses.
- **Hardware Load Balancing:** Specialised switching and routing hardware can be used to decide which server traffic is sent to. This is especially efficient if the server load is communicated between the load balancer and its connected nodes.
- **Software Load Balancing:** Load balancing software can be installed on each of the node servers or on a separate server. This solution is usually much cheaper than hardware. Commonly used options are provided by AWS, Azure or Cloudflare.

As a result of the application of load balancing, a 3-tier architecture is commonly employed for REST APIs. As shown in Figure 3.2, it consists of a client, web server and database tier. Further splitting the web server tier into web and application tier results in a 4-tier-deployment instead. Both of these architectures are easily combinable with a microservices approach. The web server application could even be split based on URIs, and only handle specific endpoints, distributed through a reverse proxy or API gateway [69, p. 322f].

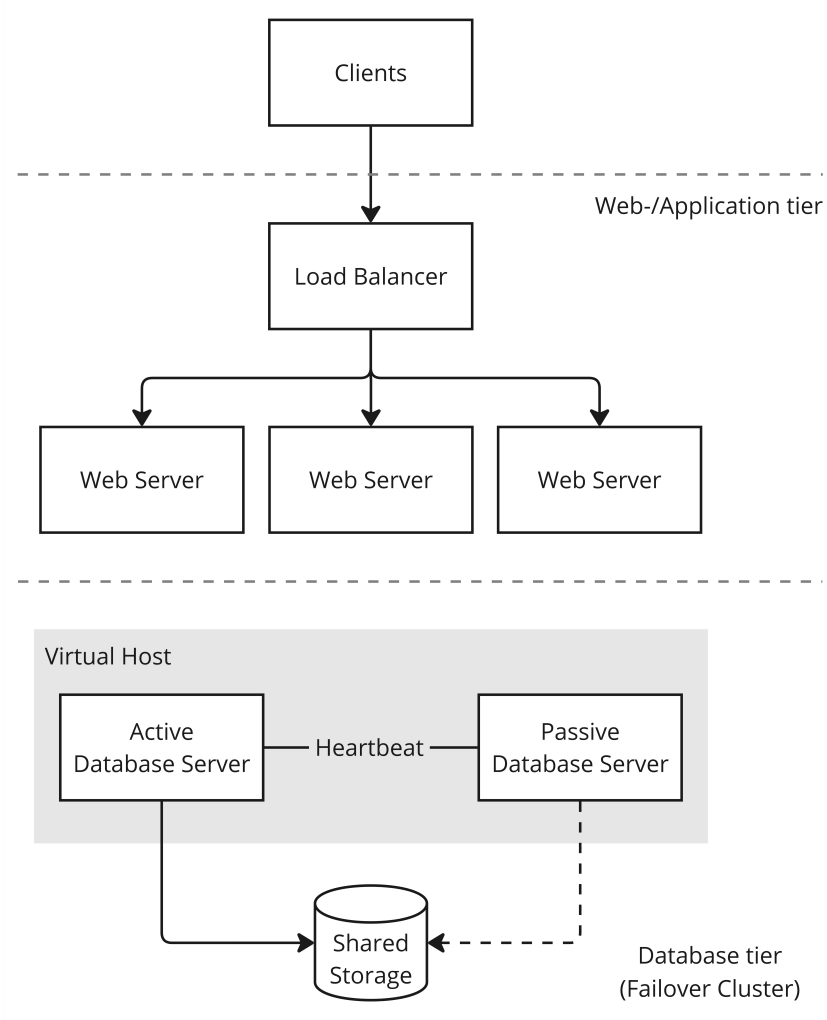


Figure 3.2: *3-tier architecture, translated from [69, p. 323]*

With vertical or horizontal scaling, the potential throughput of requests can be scaled as needed. Nonetheless, a potential for bottlenecks might still arise in different places of the system. Usually those bottlenecks are shared resources, like databases. Sharding and replication are common options to scale databases, while transactions and indexes are utilised to improve query performance. Additionally, it is recommended to paginate large result sets in API endpoints, to not cause single queries to cost too many resources [38].

While scalability is crucial, optimisations should be carefully evaluated to avoid unnecessary complexity. "Avoid premature optimisations. Scaling optimisations often come at a cost, and some of them can increase the development time of your application. Unless you have scaling problems, you probably don’t want to add that complexity" [38, p. 89]. Instead, evolving and changing an API should be based on user feedback.

3.1.8 API Gateways

As was discussed in the previous sections, the system design of a REST API should include versioning, security measures, documentation, caching and scalability measures. An option to implement all or some of these features is an API gateway.

"An API gateway is a management tool that sits at the edge of a system between a consumer and a collection of backend services and acts as a single point of entry for a defined group of APIs" [30]. As shown in Table 3.2, an API gateway covers more features than a reverse proxy or a load balancer. This does not imply that an API gateway must be integrated into REST API architecture with all of the mentioned features. Rather, developers should be aware of the requirements for their system and determine an appropriate strategy [30].

Feature	**Reverse Proxy**	**Load Balancer**	**API Gateway**
Single Backend	*	*	*
TLS/SSL	*	*	*
Multiple Backends		*	*
Service Discovery		*	*
API Composition			*
Authorisation			*
Retry Logic			*
Rate Limiting			*
Logging and Tracing			*
Circuit Breaking			*

Table 3.2: *Comparing reverse proxies, load balancers, and API gateways [30]*

As with many other software components, developers can implement an API gateway from scratch, buy a solution or use open-source options. Rather than building from scratch, it is also quite common to customise an existing solution to the desired use case.

3.2 Data Processing

As stated in chapter 2, the accompanying project for this master thesis requires data to be transformed and moved before being able to serve it to customers via a public API. In the realm of data engineering, a process like this is called Extract, Transform,

Load (ETL) process. ETL describes the workflow of "taking data from a source system, changing it to suit their [the developers] needs, and loading it to a target" [50].

3.2.1 ETL Architecture and Components

The architecture of an ETL pipeline is defined by the systems and processes involved in data extraction, transformation, and loading. A well-structured ETL architecture ensures data flows efficiently while maintaining consistency, integrity, and scalability. The design of an ETL pipeline depends on various factors, including data volume, processing latency, and target storage systems.

A typical ETL architecture consists of one or multiple data sources, the extraction, transformation and loading layers, as well as orchestration and monitoring tools. The following paragraphs outline the purpose and design principles of each of those components.

Data Sources

ETL pipelines extract data from diverse sources, such as relational databases, NoSQL stores, external APIs, and event logs (see subsection 3.3.4). The implementation of components in an ETL process is dependent on the characteristics of the source data. According to Palmer [50], these characteristics can be inquired with the following questions:

Is data bound or unbound? Is it a fixed data set that ends, or does it continuously change or grow?

What is the volume of the source data? The volume of the source data influences storage and computing costs. Higher volumes require greater efficiency to avoid latency, greater care in preventing bottlenecks, and possibly the introduction of a retention system.

How is the data structured? Unstructured data does not have a predefined schema and is often just plain text. Semi-structured data might take the form of a JSON or Extensible Markup Language (XML) payload. It "allows for flexibility in the case of changing schemas" [50], but requires data validation. Structured data is formed in a fixed schema and uses unchanging keys.

How varied is the source data? How often does it change? ETL processes should "be capable of handling disparate data types but also be flexible enough to adapt to schema changes and varying formats. [...] Failing to account for data variety can result in bottlenecks, increased latency, and a haphazard pipeline, compromising the integrity and usefulness of the ingested data." [50].

What is the quality of the source data? Data quality is defined by its accuracy, completeness, and timeliness. [50, 55].

For the Albion Online public API, multiple data sources exist, each with different premises. Some of the data is bound, or changes only infrequently, while other parts are unbound with new entries being added almost every second. Most of the data is stored in PostgreSQL databases but is partly still considered semi-structured, due to it being stored in XML or JSON format in database columns. Furthermore, some of the data is incomplete. The ETL process implementation needs to consider these premises.

Extraction Layer

The extraction phase is responsible for retrieving source data without introducing inconsistencies or system load issues. Common strategies for data extraction are:

- **Batching:** Data is read in batches instead of all at once. This is easy to implement for bound data.
- **Micro-batching:** Rather than reading big batches at low frequencies (once a day), micro batches run at high frequencies (once per minute).
- **Streaming:** Data is continuously read as it is generated. This is often supported by message services such as Apache Kafka and is more commonly used for unbound data.
- **Change Data Capture (CDC):** Rather than reading the source data, CDC tracks changes in it and updates downstream systems accordingly [50].

Transformation Layer

Once extracted, data undergoes processing to ensure it conforms to the target system's structure and requirements. According to Palmer [50], common transformation patterns include:

- **Enrichment:** Adding additional contextual information to enhance the dataset.
- **Joining:** Combining multiple data sources into a unified dataset.
- **Filtering:** Removing unnecessary or irrelevant data based on predefined conditions.
- **Structuring:** Organizing data into a defined schema or hierarchy.
- **Conversion:** Changing data formats, such as transforming text to numerical values or encoding categorical variables.
- **Aggregation:** Summarizing data by grouping and computing aggregate values like sums or averages.
- **Anonymisation:** Masking or obfuscating sensitive information to ensure privacy.
- **Splitting:** Breaking data into multiple parts for separate processing or analysis.

- **Deduplication:** Identifying and removing duplicate records to maintain data integrity. [50]

Loading Layer

The final stage involves inserting processed data into a destination system optimised for querying. This could be a data warehouse, relational database, or search index. Like data extraction, the loading process can be conducted in batches (e.g., hourly updates) or incrementally (e.g., streaming updates), depending on system requirements. [50].

Orchestration and Monitoring

ETL pipelines require orchestration to coordinate task execution, manage dependencies, and handle failures. Tools such as Apache Airflow, AWS Step Functions, or Kubernetes-based workflows provide automated scheduling and monitoring, ensuring each stage operates reliably and scales as needed. When implementing an ETL process from scratch, these features should be considered and implemented according to need. Effective monitoring allows the detection of bottlenecks, latency spikes, and data inconsistencies, improving the reliability of the ETL system.

3.2.2 Scalability and Performance

Similarly to the scalability discussion for API design in subsection 3.1.7, the development of ETL pipelines should also include measures for scalability and performance. While some techniques mentioned in the API design section can likewise be applied for data engineering, the latter field entails a few other problems and subsequent solutions.

"Scalability in data systems encompasses two main capabilities. First, scalable systems can scale up to handle significant quantities of data. [...] Second, scalable systems can scale down. Once the load spike ebbs, we should automatically remove capacity to cut costs. An elastic system can scale dynamically in response to load, ideally in an automated fashion" [55]. Developers should analyse the characteristics of the source data and apply appropriate scalability measures accordingly.

A solid understanding of the tools and frameworks used in data processing is essential for effective development. Reading the documentation and researching best practices will lead to higher quality and high-performance code. Knowing the tools well is also knowing how to identify and resolve bottlenecks[50]. Each component in the technology stack usually introduces new possible bottlenecks and consequently the need for performance improvements. Some common performance measures are listed in the following paragraphs.

Parallelisation and Concurrency

One of the ways to improve ETL performance is through parallelism. By breaking down data processing tasks into smaller, independent units, workloads can be distributed across multiple processing nodes. Parallelism can be utilised at various stages, such as extracting data from source systems, executing transformation tasks, and loading data into the target database.

Incremental processing

Reloading the entire dataset can be highly inefficient, particularly as data volumes increase. Instead, ETL pipelines should leverage incremental processing techniques. "Incremental processing [...] involves adding only new data (INSERT) or updating existing data while inserting new ones (UPSERT)" [50]. The logic for this kind of processing can be quite complex, so it may be helpful to use prebuilt tools or frameworks.

Materialisation

Materialisation improves ETL performance by precomputing and storing intermediate results, reducing the need for repeated calculations. In data pipelines, this approach reduces redundant transformations and optimises query execution by persisting transformed data in a structured format. Materialisation can be particularly beneficial when dealing with complex aggregations, joins, or computationally expensive transformations that would otherwise slow down real-time querying. Common strategies include materialised views in relational databases like PostgreSQL, where periodically refreshed snapshots of query results improve analytical workloads. However, materialisation requires careful trade-offs between storage costs and update frequency, as stale data must be refreshed periodically to maintain consistency with the source system. [50, 55]

3.3 Employed Technologies

The development of the public API and its associated services requires a selection of technologies that align with the existing infrastructure while ensuring scalability, maintainability, and performance. This section provides an overview of the primary technologies employed in the project, detailing their purpose. This preemptively gives an insight into the technology selection within the design chapter of this thesis (section 6.4). In case some of the mentioned technologies are unknown to the reader, the most important features and functions are introduced. Still, it is not possible to provide all required knowledge within the scope of this chapter.

3.3.1 Java Frameworks

The backend services in the Sandbox Interactive web infrastructure are implemented with Java 8 and a Spring Boot 2 setup with Gradle. Spring Boots's key features are simple Spring application bootstrapping, externalised configuration with profiles, and logging. At the same time, Spring itself provides dependency injection and various out-of-the-box modules like database connectors, Model-View-Controller (MVC) functionality and security configuration [2, 13].

Some older services, like the Game Info Service (section 4.1) or the Product Service at Sandbox Interactive, are implemented using the Dropwizard framework instead of Spring Boot. Dropwizard mainly distinguishes itself in how the Dependency Injection works and is comparatively small, providing faster startup times and higher throughput than Spring Boot [7].

3.3.2 REST API Frameworks and Tools

In order to develop an API in the Java ecosystem, the following tools can be employed:

springdoc-openapi automates the generation of OpenAPI specification and the corresponding Swagger UI in Spring-based projects [44].

Spring Cloud Gateway enables Spring applications to act as an API Gateway, providing security, monitoring and resiliency capabilities [12].

Postman and/or Insomnia serve as API testing tools, allowing for manual and automated validation of request/response structures and performance benchmarking [34, 52].

3.3.3 Data Processing

To support the development of an ETL process as defined in section 6.6 various tools can be utilised. The following list describes the purpose and functionalities of Java libraries relevant to data processing.

Jakarta Persistence API (JPA) is a specification for Object-Relational Mapping (ORM) that "defines how to persist data in Java applications" [4].

Hibernate is an implementation of the JPA specification that is widely used and supported [4].

Spring Batch is used for batch processing tasks. It provides robust transaction management, retry mechanisms, and job scheduling capabilities. It is commonly used to implement ETL processes in Java [11].

MapStruct provides code generation for object mapping functionality [73]. It simplifies the transformation between Plain Old Java Objects (POJOs) like Data Transfer Objects (DTOs) and JPA entities.

3.3.4 Data Storage

In order to store the processed data in the last step of an ETL process, a data storage solution needs to be selected. Currently, PostgreSQL and Elasticsearch are both in use for similar use cases at Sandbox Interactive.

"PostgreSQL is a powerful, open source object-relational database system with over 35 years of active development that has earned it a strong reputation for reliability, feature robustness, and performance" [75]. Elasticsearch on the other hand "is an open source distributed, RESTful search and analytics engine, scalable data store, and vector database capable of addressing a growing number of use cases" [22]. Using these two technologies as representatives, the following paragraphs outline the advantages, drawbacks and resulting use cases of relational databases and document-based data storage.

Relational Databases

"A relational database is a type of database that stores and provides access to data points that are related to one another. Relational databases are based on the relational model, an intuitive, straightforward way of representing data in tables. In a relational database, each row in the table is a record with a unique ID called the key. The columns of the table hold attributes of the data, and each record usually has a value for each attribute, making it easy to establish the relationships among data points" [46]. According to [28, 33, 46] relational databases provide the following benefits:

- **ACID Compliance:** Transactions in relational databases provide Atomicity, Consistency, Isolation and Durability, also referred to as ACID properties.
  - Atomicity ensures that all operations within a transaction are completed as a single unit—either all succeed, or none are applied.
  - Consistency guarantees that the database remains in a valid state before and after transactions.
  - Isolation ensures that concurrent transactions do not interfere with each other, maintaining data integrity.
  - Durability warrants that once a transaction is committed, its changes are permanent, even in case of a system failure.
- **Ease of Use:** Relational databases commonly use Structured Query Language (SQL) to execute even complex queries or transformations.

- **Reduced Redundancy:** Database normalisation and stored procedures are common practices in relational databases that help eliminate redundancy of data and processes.
- **Backups and Disaster Recovery:** The majority of relational database management systems offer simple export and import functionality, which is useful for failure recovery.
- **Collaboration:** The isolation of database transactions facilitates collaboration on data access and processing. Included locking mechanisms ensure data integrity for this case.

Therefore, the "relational model is used by organisations of all types and sizes for a broad variety of information needs. Relational databases are used to track inventories, process ecommerce transactions, manage huge amounts of mission-critical customer information, and much more. A relational database can be considered for any information need in which data points relate to each other and must be managed in a secure, rules-based, consistent way." [46]

PostgreSQL goes beyond these characteristics and is labelled as an Object-Relational Database Management System (ORDBMS). It extends the traditional relational database model by integrating object-oriented features such as user-defined types and table inheritance [74].

In the Sandbox Interactive web team, PostgreSQL databases are the standard solution for structured data storage.

Document-Based Data Storage

Document-based storage systems, such as Elasticsearch, adopt a schema-less approach, typically storing data in JSON format for flexible querying and indexing. They are categorised as non-relational or NoSQL databases, which means that data queries cannot be done through SQL and often require the usage of a custom query language. Due to their flexibility, "Document databases are often used in scenarios where data is diverse and changes over time" [54].

In contrast to relational databases, document-based databases can be scaled horizontally by adding more servers. The ability to scale horizontally renders them a preferred option for applications dealing with large volumes of data [54].

Additionally, document databases commonly feature a model called "eventual consistency". They do not comply with ACID requirements and rather prioritise data availability over correctness [54].

Elasticsearch, in particular, provides strong search capabilities on top of these features. It employs an inverted index and tokenisation techniques, enabling efficient full-text search, filtering, and ranking [22].

At Sandbox Interactive, Elasticsearch is mainly utilised as part of an Elasticsearch, Logstash, Kibana (ELK) stack to store game and web logs (see section 4.3). Other Elasticsearch clusters are employed in each data center to store and prepare game data for the Game Info Service. This is further explained in Figure 4.3.

3.3.5 Infrastructure and Deployment

The following technologies support operational requirements for the development of a system, like deployment, monitoring, and logging:

Jenkins is an open-source automation server that can be configured through a web interface or scripting [37].

Puppet is an engine for automated administration of Linux, Unix or Windows systems based on a centralised specification [53].

Datadog provides a monitoring platform by tracking custom metrics and providing information on application performance, infrastructure and network activity [19].

Logstash forms the backbone of log aggregation in an ELK stack, allowing for efficient indexing, searching, and analysis of system logs [23].

This concludes the overview of technologies employed in this project and, thereby, the discussion of fundamental knowledge for this thesis. This chapter explored the industry standards and best practices for API development and data processing, as well as the expected technology stack, providing the necessary groundwork for the subsequent design and implementation of the Albion Online Public API.

4 Analysis of the Current System

4.1 The Game Info Service

An existing system called the Game Info Service (GIS) currently provides access to game data via REST API. The GIS has been developed over four years and functions as a monolithic system for data processing, search, and delivery. Currently, it is used to retrieve data for the Character Builder[1] and the Killboard[2] on the official Albion Online website. Data retrieval is handled via JavaScript in the frontend instead of the backend. As a result, the GIS API was unintentionally exposed. Since users have discovered this, they have been using the exposed endpoints without any form of authentication, authorisation, request throttling or rate limiting. This is a security gap that is intended to be closed once an intentional public API can replace the GIS.

4.1.1 Included Features

Currently, the GIS API includes the following endpoints:

- Recent and most influential kills
- Recent and upcoming Guild versus Guild (GvG) events (The GvG feature in the game is not active anymore)
- Recent and most influential battles
- Recent arena/crystal league matches (PvP matches in Albion Online)
- Top players and guilds in crystal league
- Player, guild, alliance and item information
- Item and spell icon rendering
- Top guild members by PvE-, Gathering-, and Crafting-Fame (this feature was disabled due to performance issues)
- Top players, ranked by PvP-Fame, can be filtered by selected time period and used weapon (also disabled due to performance issues)
- Store or request player builds[3]

[1] albiononline.com/characterbuilder
[2] albiononline.com/killboard
[3] For further explanation of a few of these features, check the glossary

Additionally, code was written for an endpoint that returns information on clusters, which are the zones making up the game world, each roughly 1x1 kilometre in size [60]. It is not fully exposed, so it remains unused.

Most of these endpoints are JSON based, only the rendering endpoints deliver a PNG response. Similarly, most endpoints can be reached via `GET` requests and are intended for data delivery. Only the endpoint for player builds allows a `POST` request to store new data.

In terms of RESTfulness, the current API therefore barely reaches level two of the Richardson Maturity Model (see subsection 3.1.1). A documentation of the API was never provided, not complying with one of the best practices for API development, as described in subsection 3.1.5. Instead, the available endpoints are documented by community members who have discovered them, for example on tools4albion.com/api_info.php.

Apart from the lack of documentation, the provided endpoints break with other REST API Design Principles (subsection 3.1.2) as well. Some URIs use uppercase letters and underscores, for example `/items/_weaponCategories`, which should be `/items/weapon-categories` according to the guidelines by Masse [41]. Other URIs also break with the convention that a forward slash separator (/) should indicate a hierarchical relationship [41, p. 11]. An example of this is `/events/playerweaponfame`, which returns the "players with [the] most kill Fame within the selected time range and weapon category" [24]. Rather than providing an underlying set of information of "events", the `playerweaponfame` path element acts as a selector for sorting. According to the guidelines, the endpoint should rather be available through `/players?sort=weapon-fame&time-range=your-time-range`.

Additionally, the endpoints for player builds do not comply with the REST principle of integrating with a uniform interface due to the fact that the `POST` method can be used to create, update and delete builds. Instead, the application should provide a `PUT` and `DELETE` endpoint alongside the `GET` endpoint.

Apart from the REST API, the GIS also provides a feature that detects battles in the game. For this the GIS utilises guild and alliance affiliation of the involved players. The implemented logic is not tested or documented anywhere. Further research and testing need to be done should a similar feature be implemented in the new systems.

4.1.2 Usage

The mentioned endpoints in the GIS were mainly implemented to be used for tooling on the Albion Online website, namely the Killboard and the Character Builder.

The Killboard (shown in Figure 4.1) shows recent kills and battles that happened in the game, as well as the top kills in the current and previous week or month. It uses the rendering endpoints to visualise the equipment and inventories of the participating

players and provides a frontend for most of the information that the kills endpoints deliver.

Figure 4.1: *Anonymised interface of the Killboard showing the "Top PvP Kill this month" on the Albion Online website*

The Character Builder (shown in Figure 4.2) is a tool that provides players with the option to exchange information on their player builds. The tool provides a graphic interface to create new builds and set a name, a richtext-based description, category tags, and the required gear for the build. Users can then filter, view, comment and vote on those kills. While the main build information is stored via the GIS, the website uses a different connected database to store comments and votes.

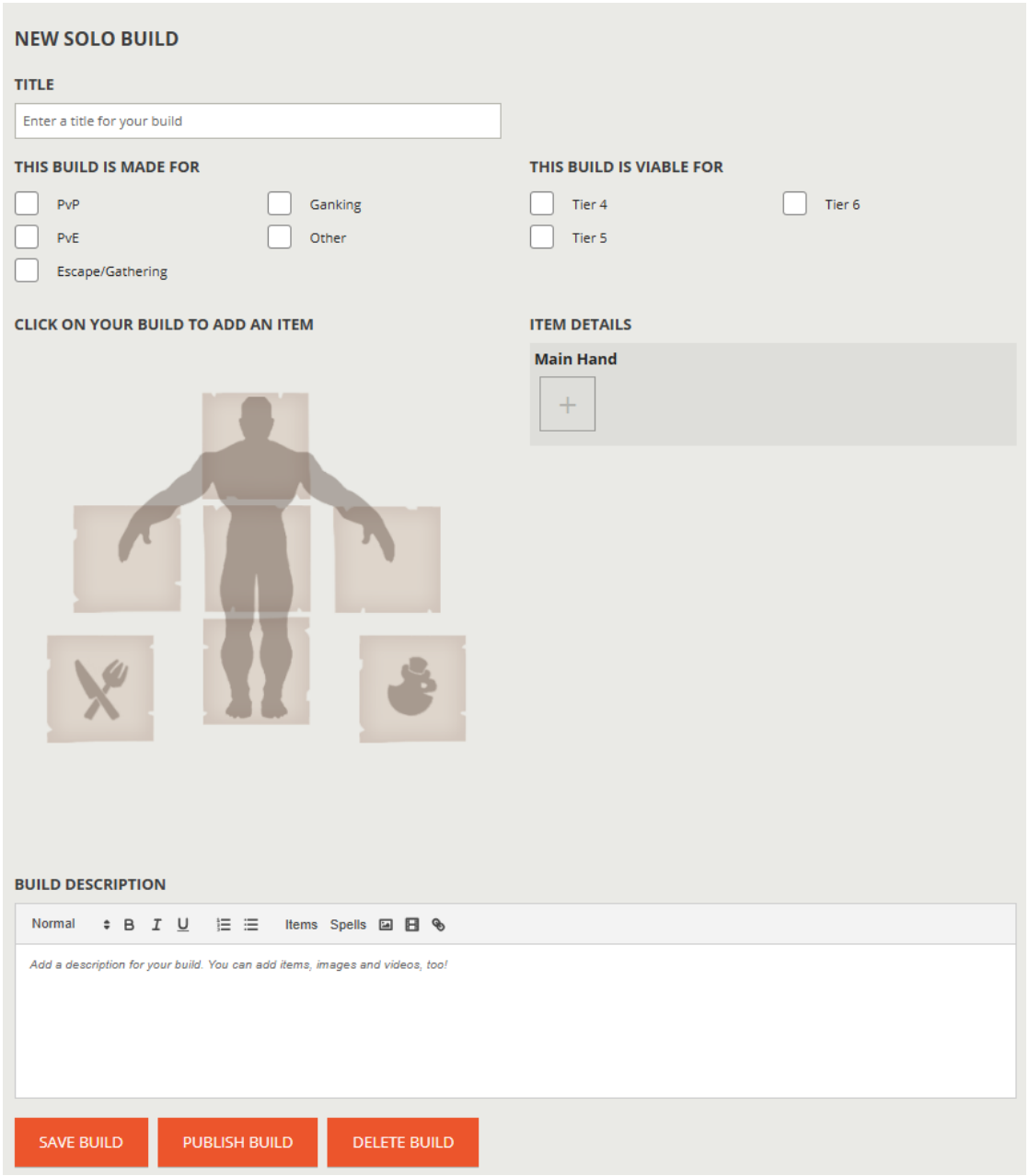


Figure 4.2: *Interface to create a new Solo Build in the Character Builder on the Albion Online website*

Apart from these tools provided by Sandbox Interactive, third parties are using the GIS endpoints as well, and have created their own Killboards or other visualisations and tools. Table 4.1 lists known third-party API clients and their functionalities. Note that this only includes clients and functionalities that consume the GIS API. Other third-party tools will be further discussed in section 4.2

Tool Name	Features
albiononlinetools.com	Player and Guild Statistics, Random Build Generator, Killboard
albiononline2d.com	Kill Analytics and Player Statistics, link kills to Twitch VODs
tools4albion.com	Alliance and Guild History
albionfreemarket.com	Determines most effective Items for PvP, by tracking which items are most used in kills and yield the highest Fame amounts
albionbattles.com	Battle Reports
avaloniantool.com	Search Tool for Item and Spell Icons
albiononlinegrind.com	Character Builder (using their own data storage for builds)
metabattle.com/albion	Character Builder (using their own data storage for builds)

Table 4.1: *Third-Party consumers of the Game Info Service API and their features*

When analysing the third-party consumers of the GIS, it becomes apparent that only few of them query the GIS API directly. Rather, they appear to have their own databases set up that are filled with information once gathered from the GIS, and are designed to query the relevant information for the specific tools. As an example, albiononline2d.com provides data on the "Top Victims by fame (7 days)" which cannot be queried from the GIS. The lack of filter and sorting options when querying information from the GIS and the high response times (see subsection 4.1.4) are likely the reason that third-party developers have chosen this route.

It also becomes apparent that the community is not completely satisfied with the provided tools on the Albion website, building their own Killboards or Character Builder. This further supports the claim made in subsection 2.2.1, that official game APIs enable game studios to leverage third-party development.

Generally, the analysis of the GIS usage also shows which information is relevant to the community and can, therefore, guide the design of the new API. Following this logic, the endpoints for spell and item rendering, kills, battles and player, guild and alliance statistics should all be included in the new implementation. There, they should be enhanced to allow further sorting and filtering options.

4.1.3 Tech Stack and Architecture

The GIS is currently set up as a Java 8 application using the Dropwizard framework, as opposed to most of the remaining web services at Sandbox Interactive GmbH, which use

Spring Boot 2 instead. As shown in Figure 4.3, game servers write relevant information into the `dataexport` PostgreSQL databases as JSON-formatted text. The GIS in each environment regularly processes that data and writes it into a connected Elasticsearch cluster, which means that it can be considered an ETL Service (see section 3.2). The Elasticsearch clusters act as document-based databases and search tools to retrieve necessary data when handling external requests to the REST API endpoints in the GIS.

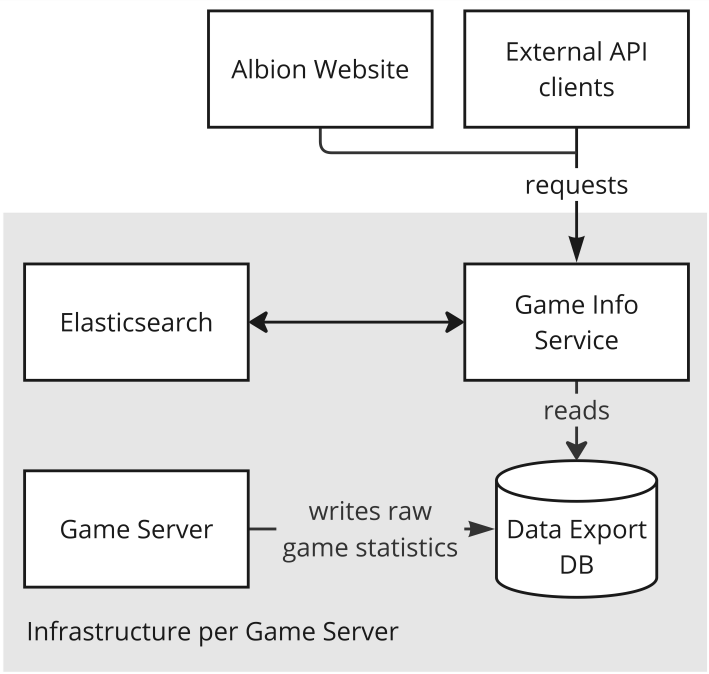


Figure 4.3: *Current architecture surrounding the GIS*

4.1.4 Resource Usage and Performance Metrics

Currently, the GIS infrastructure costs Sandbox Interactive about 2,500€ per month. The Elasticsearch cluster in the US currently runs on five connected machines, with a combined amount of 5 TB of storage. Since the data in the Elasticsearch constantly grows, so does the disk size of the corresponding machines. Because the servers in the US are hosted on bare metal machines, it has become difficult to increase the available storage. Instead, new machines had to be purchased and set up to join the Elasticsearch cluster when the storage threatened to run full. While storage is not expensive, the compute required to support Elasticsearch is.

A recent unexplained issue in the GIS caused the kill processing to not work, which went unresolved for ten days. When the issue was finally resolved, it took over two days for the GIS to catch back up.

As shown in Figure 4.4, the GIS API has received an average of 4000 requests per minute in 2024. In comparison, all pages of the official Albion Online website have

received an average of around 1000 requests per minute, which is proof that the GIS is being used by anonymous and unauthorised clients.

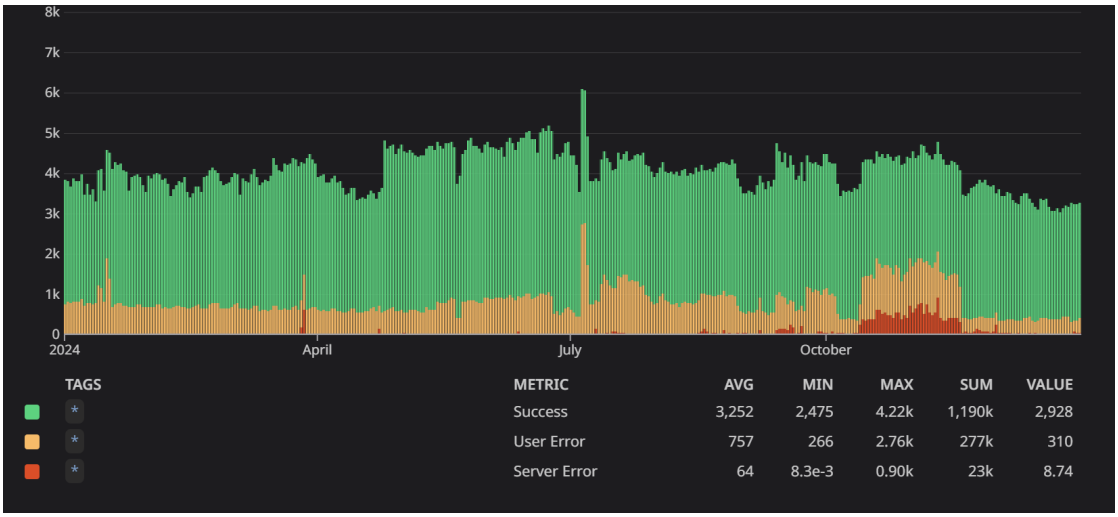


Figure 4.4: *Amount of requests per minute against all the GIS; Exported from Datadog*

As Figure 4.5 shows, the oldest instance of the GIS (the GIS on the american server) has an average response time of 9.13 seconds throughout 2024. This is above the limit of 1 second to keep a user in their flow of thought and is very close to the limit of 10 seconds for keeping the user's attention, as described in [45, p. 135].

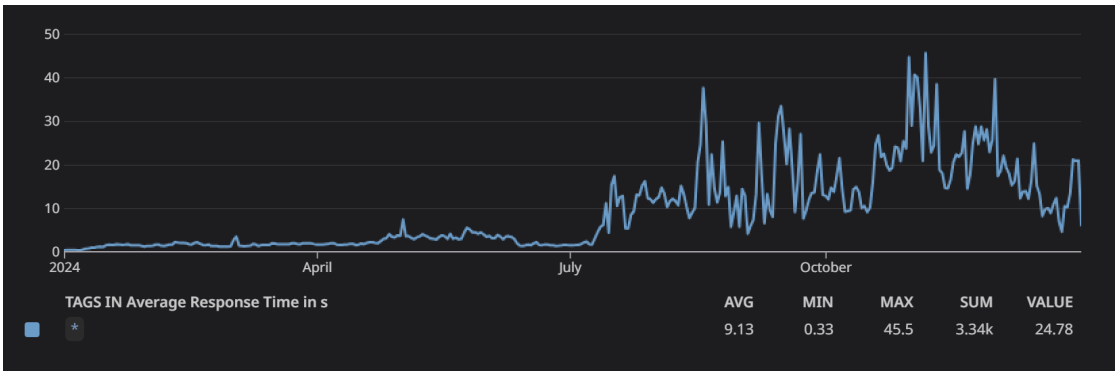


Figure 4.5: *Average response times (in seconds) of the GIS in the American environment; Exported from Datadog*

Additionally, the GIS on the American servers usually takes around 30 minutes to restart, which results in an equally long downtime on the accompanying REST API.

4.1.5 Maintainability

Given the performance issues and lack of security measures, the question arises whether the current setup can be maintained and, specifically in this case, altered and improved

upon. According to ISO Standard 25010 [35], software maintainability can be defined through the following characteristics:

- Modularity - Degree to which a system or computer program is composed of discrete components such that a change to one component has minimal impact on other components.
- Reusability - Degree to which a product can be used as an asset in more than one system, or in building other assets.
- Analysability - Degree of effectiveness and efficiency with which it is possible to assess the impact on a product or system of an intended change to one or more of its parts, to diagnose a product for deficiencies or causes of failures, or to identify parts to be modified.
- Modifiability - Degree to which a product or system can be effectively and efficiently modified without introducing defects or degrading existing product quality.
- Testability - Degree of effectiveness and efficiency with which test criteria can be established for a system, product or component and tests can be performed to determine whether those criteria have been met. [35]

As noted in subsection 4.1.3, the GIS contains data processing, data search (Elasticsearch) and data delivery (REST API) components. These components are heavily intertwined and could currently not be extracted as independent modules. The application can, therefore, not be described as modular.

Due to the framework choice (Dropwizard), none of the code in the GIS has been reused so far. Arguably, some of the logic tied to game design decisions can and should be reused for this project. Due to the low modularity, it will be difficult to extract said logic, therefore lowering the reusability of the GIS code.

Moreover, the GIS does not provide good analysability either. Internal logging conventions have not been applied at all, and the service has an output of over 400 logs per minute, increasing the difficulty of analysing arising failures. There is no documentation provided on the architecture of the service, requiring developers to understand the structure of the application by going through the code. With the code spanning multiple layers of logic and functionality, this is also a difficult task.

In addition, the modifiability of the GIS is also considered low due to various reasons. The documentation did not provide sufficient guidance for setting up the service locally. Assistance from multiple departments was required, including a hardware upgrade from 32 GB to 64 GB RAM. With this time and these resources not available to everyone in the development team, developing and testing locally is close to impossible. Additionally, the integration with Elasticsearch as a data aggregation and search tool is strongly embedded into the code and can not be exchanged for a different data storage

option easily. Furthermore, a lot of configurations and strings are hard-coded, which increases the risk of breaking (parts of) the application, even for smaller changes.

Lastly, the testability of the GIS is considered low as well. No integration tests have been written, but a few unit tests, which provide a test coverage of 6%. On top of that, there is little to no documentation on the functionality implemented in the GIS, consequently increasing the difficulty to reliably test the GIS outside of automated testing.

In conclusion, the currently implemented solution is considered difficult to maintain.

4.1.6 Backward Compatibility

Another point to consider when analysing the current solution is its backward compatibility. In the past, the GIS has encountered defects when changes were made to the `dataexport` database structure or when new items and spells were introduced to the game world. There is no established process or documentation to adhere to for these cases.

When accessing an old character build from the GIS, the user is also very likely to receive incorrect information. There is no historical information on items and spells, therefore an old build will return with current values and stats, instead of those from the point in time where the build was created.

4.1.7 Lessons Learned

The analysis highlights several key lessons for future implementations. Firstly, the practice of making frontend calls directly to the API from the official website should be discontinued, as this has led to unintended API exposure. Instead, all data requests should be properly routed through backend services.

The architectural approach should shift from a monolithic structure to a microservice architecture, adhering to the Single responsibility principle. This would improve modularity and, in return, the maintainability of the new solution. Additionally, the current document-based data storage system has shown scalability and performance limitations, suggesting a need for an alternative storage solution.

Security measures need significant enhancement. This includes implementing proper authentication mechanisms to prevent anonymous calls, as well as introducing request throttling and rate limiting to prevent system abuse. The system should also align with internal conventions and utilise the company's standard tech stack to ensure consistency across services.

Proper monitoring systems should be established to track performance and usage metrics effectively. Furthermore, comprehensive documentation is essential, covering local setup procedures, architectural decisions, and detailed functionality descriptions.

This should be complemented by thorough unit and integration testing to ensure system reliability.

Lastly, a versioning system for static game data, including spells and items, should be implemented to maintain historical accuracy and prevent issues with backward compatibility.

4.2 Third-Party Solutions

Apart from the GIS API discussed in the previous section, another solution for game data access has been set up by third-party developers: The Albion Online Data Project. According to their website, their goal is "to collect and distribute realtime information for Albion Online. This is achieved with a downloadable client that monitors network traffic specifically for Albion Online, identifies the relevant information, and then ships it off to a central server which distributes the information to anyone who wants it" [72].

The project provides a REST API that provides current and historical item prices, as well as gold prices, therefore mainly covering the use case of marketplace information. The API can be accessed anonymously and is rate limited per IP to 180 requests per minute and 300 requests in five minutes [72].

Many of the tools already examined in section 4.2 provide functionality built on top of the Albion Online Data Project, for example crafting cost or farm profit calculators [72].

While the Albion Online Data Project is known to Sandbox Interactive and its practices are officially tolerated [77], this project aims to provide users with a more secure and reliable way of accessing market-related data.

4.3 Available Infrastructure and Tools

The development process outlined by this thesis is strongly supported by established infrastructure and tooling at Sandbox Interactive. This section provides an overview of the available components and tools relevant to this project.

Reverse Proxy and API Gateway

All external communication with Albion Online's services is routed through an Nginx reverse proxy. That way, almost all incoming requests are sent to the "revproxy server" and forwarded to configured internal upstreams from there.

Additionally, an API Gateway with basic functionality has already been implemented and deployed to the production system. It is primarily used to provide authorised customer support agents access to player data. External communication with the API Gateway also goes through the Nginx reverse proxy.

Logging and Monitoring

To facilitate real-time monitoring, log aggregation, and system diagnostics, the infrastructure utilises an ELK stack. Complementing this, Datadog is used for monitoring and alerting.

Builds and Deployments

Automated build and deployment pipelines are established using Jenkins. These pipelines handle the compilation, testing, and deployment of services. The servers that are used for the web infrastructure are configured and managed through Puppet. This includes network and nginx configuration, Datadog connectivity, database setups and user management.

Service Architecture

The backend services developed by the web team at Sandbox Interactive follow a modularised Spring Boot-based architecture, which adheres to a structured division into `client`, `common`, and `server` modules. The `server` module serves as the runnable application, whereas the `client` module acts as an API wrapper library, to be used by other services that depend on the provided functionality. The `common` module is utilised in both other modules and usually includes DTOs or enums.

A service template is provided for Java applications like these (see section 7.1), that includes Gradle setups with the required dependencies for each module. The `server` module includes libraries that automatically establish the logging setup required for the ELK stack and the connection to Datadog. The template also ensures test execution in Jenkins on a Git push.

This existing infrastructure provides a proven and stable foundation for the implementation of the public API, as well as tooling that can be reused to simplify common processes.

5 Requirements

With the analysis of the current web infrastructure at Sandbox Interactive in the previous chapter, and the research on API design best practices in section 3.1 completed, it is possible to define the requirements for the Albion Online Public API. This chapter provides an overview of what features the API should include as functional requirements, and how the API should perform as non-functional requirements. Technical, business and legal constraints are discussed thereafter, as well as assumptions that will guide the design and technical decisions. Lastly, this chapter prioritises the requirements and defines the scope to be fulfilled as part of this master's project.

Requirements and constraints are defined by the stakeholders for the project, which in this case are community members, third-party developers, the game designers for Albion Online and the development team at Sandbox Interactive. The requirements in the following sections were elicited through informal internal stakeholder interviews, and through the analyses of user interests in section 2.2 and chapter 4. There were no formal requirements for the engineering process. Conversely, the author of this thesis served as owner for the project, and has therefore defined the requirements herself, in collaboration with relevant stakeholders in the company.

5.1 Functional Requirements

Functional requirements define the capabilities that a system should hold. They "describe the services the system should provide and how the system will react to its inputs" [49]. On a high level, the Albion Online API is required to meet the needs of the community and, more specifically, of third-party developers while remaining in line with the overall vision for the game.

ID	Description	Rationale	Dependencies
FR-1	The API MUST provide access to game data, including kills, marketplace data, static game data, and player, guild and alliance statistics. Data SHOULD be retrievable through filtering and sorting mechanisms.	This is the main use case of an official game API.	Database schema design, data processing and transfer
FR-2	The API MUST require authentication for all endpoints. User authorisation MUST be scoped.	Protects sensitive data, enables third-party development and FR-3	Account management
FR-3	The API MUST enforce request rate limits. Clients exceeding rate limits SHOULD receive an HTTP 429 response.	Protects system availability by preventing excessive load from misconfigured or malicious clients.	API gateway configuration
FR-4	The systems SHOULD be equipped with Datadog metrics, monitors and logging capabilities.	Facilitates debugging, availability measures, and performance optimisation.	Datadog integration, Logstash
FR-5	The API MUST support versioning to accommodate future updates without breaking existing client integrations.	Enables iterative improvements while maintaining backward compatibility.	API gateway configuration
FR-6	The API SHOULD implement caching mechanisms	Improves performance and responsiveness under high-load conditions.	Redis, cache invalidation policies
FR-7	The API MUST support pagination for endpoints returning large result sets.	Prevents excessive data transfer in a single request, improves overall performance.	Database query optimisation

Table 5.1: *Functional Requirements of the Public API, Part 1*

ID	Description	Rationale	Dependencies
FR-8	The API MUST return standardised error messages and HTTP status codes.	Ensures compliance with REST	Endpoint design
FR-9	Clear and up-to-date API documentation MUST be provided to external users, including authentication methods, rate limits, and endpoint specifications.	Facilitates third-party adoption and reduces support overhead.	API documentation tools
FR-10	Clear and up-to-date technical documentation SHOULD be provided to internal developers for all services, including the software architecture, features and setup instructions	Facilitates system maintainability and extensibility	GitLab, Confluence
FR-11	The API MUST use secure communication protocols	Encrypts communication to prevent data interception.	TLS/SSL certificate

Table 5.2: *Functional Requirements of the Public API, Part 2*

5.2 Non-Functional Requirements

Non-functional requirements describe "how the system behaves concerning some observable attributes like reliability, reusability, maintainability" [49].

ID	Description	Rationale	Dependencies
NFR-1	The API MUST process requests within a response time of 1s or less for 95% of requests.	Provides a smooth user experience.	Query optimisation, caching
NFR-2	The systems MUST be extensible for more use cases	Simplifies new development	None

Table 5.3: *Non-Functional Requirements of the Public API, Part 1*

ID	Description	Rationale	Dependencies
NFR-3	The API SHOULD maintain an uptime of at least 99%.	Ensures high availability and reliability for third-party applications and users.	Failover mechanisms
NFR-4	The system MUST be able to scale horizontally.	Support increasing user demand without performance degradation.	Stateless architecture
NFR-5	The API MUST follow industry security best practices, including input validation, authentication enforcement, and protection against OWASP Top 10 vulnerabilities.	Reduces attack surface and mitigates risks of common security threats.	API gateway configuration, penetration testing
NFR-6	The API and its documentation SHOULD support localisation where applicable	Expands accessibility and adoption across different regions.	Localisation frameworks
NFR-7	The API SHOULD comply with REST principles and reach level two of of the Richardson Maturity Model.	Improves usability and software quality	None
NFR-8	The API endpoints SHOULD follow best practices in REST API design, as discussed in subsection 3.1.2.	Improves usability and complies with industry standards	None
NFR-9	The API SHOULD ensure that retrieved data is correct and up-to-date.	Provides an improvement over the GIS	Performant and correct ETL process

Table 5.4: *Non-Functional Requirements of the Public API, Part 2*

ID	Description	Rationale	Dependencies
NFR-10	The ETL process SHOULD process historical data within a maximum of 5 days.	Allows data recovery within a working week	Performant ETL process
NFR-11	All systems SHOULD be maintainable according to ISO Standard 25010.	Enables developers to maintain the system long-term	None

Table 5.5: *Non-Functional Requirements of the Public API, Part 3*

5.3 Constraints

The design and development of a public API for Albion Online is subject to various technical, business and legal constraints. These are usually seen as non-functional requirements [49], but are listed separately here for readability.

ID	Description	Rationale	Dependencies
C-1	The API MUST comply with General Data Protection Regulation (GDPR)[2] and other applicable data protection regulations.	Ensures legal compliance and protection of user data.	Data anonymisation, secure storage policies
C-2	The API MUST only expose data that was approved by the game design team.	Prevents unfair advantages for players using third-party tools.	None
C-3	The systems MUST integrate with the existing game infrastructure without requiring major architectural changes.	Avoids significant refactoring costs and ensures compatibility with current systems.	ETL process
C-4	The API SHOULD integrate with the existing web infrastructure without requiring major architectural changes	Enables usage of existent tooling and internal libraries	None

Table 5.6: *Constraints of the Public API*

5.4 Assumptions

All of the listed requirements are based on a few assumptions, regarding the expected usage of the API. Firstly, it is assumed that the majority of API consumers besides the Albion Online website will be third-party developers. Therefore the API should be designed with usability and ease of integration for third-party development in mind. This user group is, therefore, assumed to be capable of reading technical documentation.

Secondly, the API is expected to experience fluctuating load. Additionally, the GIS API has a history of being attacked through DDoS, which is assumed to happen for the new API as well. The system should, therefore, be designed to handle load spikes and potential attacks without significant degradation to the performance of its own services or other infrastructure.

Lastly, it is assumed that the data structures provided by the game are subject to change. With new game content and balancing changes, the item or spell structure might, for example, change. This assumption suggests the need for a versioning system for the API and the underlying data.

5.5 Prioritisation and Scope

The requirements for a public API for Albion Online far exceed the scope of a master's project. The system and API design can be completed in advance and will be discussed within this thesis. However, the accompanying project will implement a Minimum Viable Product (MVP), rather than all the requirements and desired use cases. The MVP intends to pave the way for a full-fletched public API.

For a MVP, the requirements marked with "MUST" will be implemented first, since they have a higher priority. Requirements marked with "SHOULD" can also be implemented within the scope of this thesis, but are not obligatory. The main restriction will lie in a modification of FR-1. Rather than requiring the API to provide access to all kinds of game data, the new requirement calls for the following:

FR-1-MVP: The API MUST provide access to kill data.

6 Architecture and Design

This chapter provides the generation and explanation of the design for the Albion Online Public API and its infrastructure. It details the decisions made to ensure the API fulfils all of the requirements laid out in the previous chapter and meets performance, security, and usability standards while accommodating future growth and potential changes. It breaks down the steps that were taken and modifications that were made to initial plans, providing reasoning for each. The full design includes a system architecture plan, API design as outlined in section 3.1, necessary user interface mockups and the selection of technologies and frameworks. Additionally, it encompasses the design of the database schema and a concept of the data processing pipeline.

6.1 System Architecture

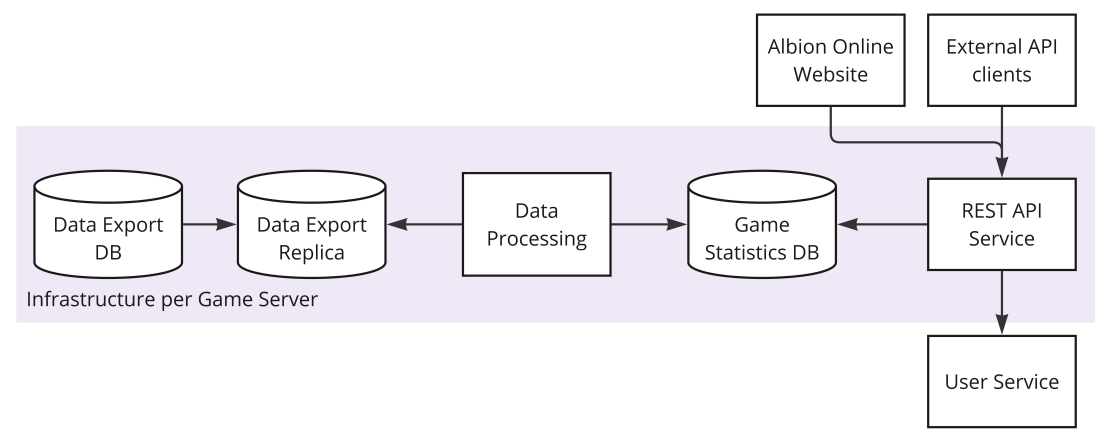


Figure 6.1: *Initial architecture plan for the API systems*

Figure 6.1 shows the initial architecture plan that was made. The `dataexport` database is replicated, and a data processor reads data from that replica. The processed data is then written into a new "Statistics" database. From there, the REST API Service can query the data required to respond to any incoming requests. For authentication and authorisation, the REST API Service communicates with the User Service. Except for the User Service, all mentioned components of this infrastructure are deployed in each

datacenter.

This approach has two advantages: Firstly, if there is increased load on theAPI, it will never have any impact on the game or game adjacent systems. Secondly, the new Statistics database can be shaped to the needs of the REST API, and will be adjustable to any changes or new use cases.

One disadvantage of this approach is the latency directly within API calls. The REST API Services in Singapore and Amsterdam would have significantly slower response times because there is latency between them and the User Service in the US. The Albion Online Website is deployed in the US as well, so calling services in the other data centers would produce another latency point.

Another drawback of the initial design is that the REST API Service has too many responsibilities. It would have to handle authentication, authorisation, rate limiting, data access and caching. As was discussed in subsection 3.1.8, a common solution to this is an API gateway.

Additionally, putting more load on the User Service for API authentication and authorisation is a risk that should be circumvented. The User Service is deemed as one of the most critical components in the web infrastructure, and already receives a lot of load. The code for the User Service is also already quite bloated, and adding more features may decrease the maintainability.

When discussing this architecture with co-workers at Sandbox Interactive, it was also noted that the usage of the `dataexport` replica was not necessary. While the benefit could have been even less load on game adjacent systems, the current load on the `dataexport` database is already very low. Instead, the data processing service can be configured to not create too much load and, in return, the replication lag will not have any impact on data availability.

Iterating on this first draft leads to the system design shown in Figure 6.2. Here, the advantages from the previous design are still incorporated, while the flaws and obsolete features are removed.

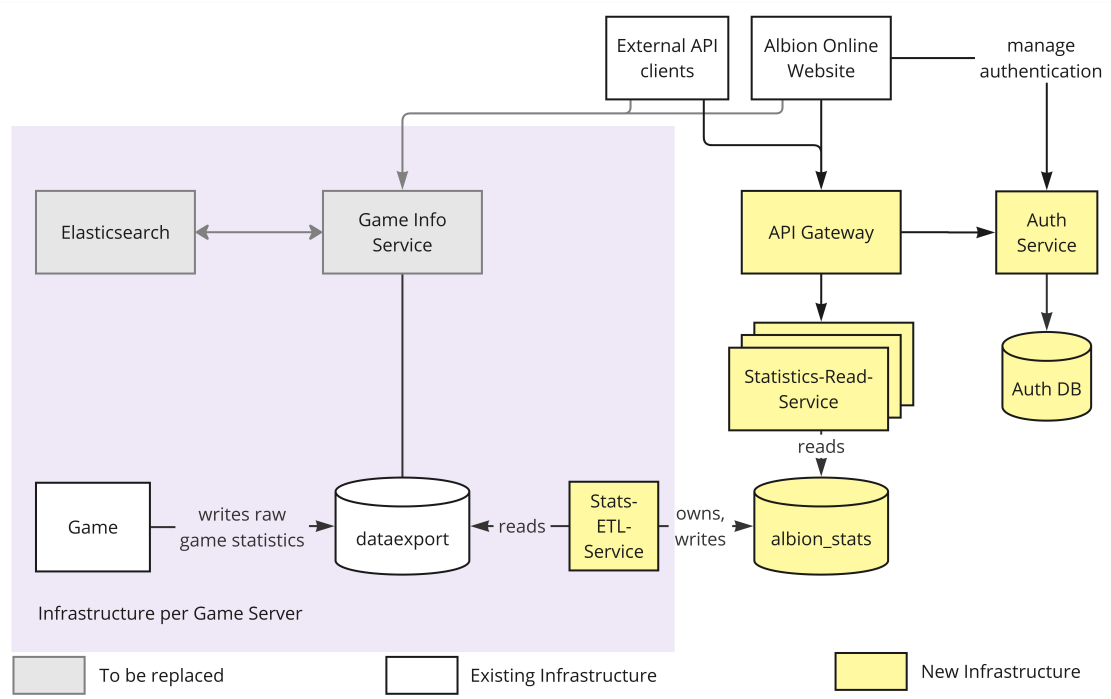


Figure 6.2: *Architecture Diagram*

The planned architecture for this project now introduces several new components. The Stats-ETL-Service reads raw data from the `dataexport` database, transforms it into a structured format, and stores it in the new `albion_stats` database.

External API requests are routed through an API Gateway, which acts as a central access point for both external clients and the Albion Online website. From there, requests are distributed to the Statistics-Read-Service instances. The service then retrieves the requested data from `albion_stats`.

The new Auth Service manages authentication and authorisation matters, with credentials and access scopes stored in the corresponding `auth_service` database.

With this, the latency between data centers now only concerns the data processing pipeline. This is possibly an issue when trying to comply with NFR-10: "The ETL process SHOULD process historical data within a maximum of 5 days." After discussion with various programmers at Sandbox Interactive, it was decided that the issue should be dealt with once it arises though, and that the benefit of having no latency in API calls outweighs it.

Additionally, the API service is split into the Statistics Read Service, and the API Gateway, where the API Gateway should be responsible for authentication, authorisation and rate limiting. After applying all these measures, the gateway passes requests through to the Statistics Read Service, whose purpose is to query, potentially process and return the requested data.

Furthermore, rather than communicating with the User Service for authentication

and authorisation purposes, an Auth Service is implemented, that owns and queries a new database that stores authentication information. With this, communication with the User Service is not necessary at all, so that no additional load is put onto a critical system.

In this architecture, all services are designed to be stateless, allowing horizontal scaling on every layer.

6.2 API and Endpoint Design

As was already discussed in subsection 3.1.1, the Albion Online Public API is supposed to be a REST API using HTTP. Further design decisions have already been made in chapter 5 as well. This section will describe the API design in more detail, including security measures to be taken, rate limiting policies, API versioning techniques and endpoint design. Most of the upcoming measures are applicable to all planned use cases, whereas the endpoint design is restricted to the "kills" use case, as recorded in FR-1-MVP.

6.2.1 Security

Among the various authentication methods available (see subsection 3.1.4), API keys and OAuth 2.0 have emerged as the dominant approaches (see Table 2.2). Each of these methods offers distinct advantages and trade-offs, making the choice highly dependent on the specific requirements and constraints of the system.

OAuth 2.0 is widely used for third-party authentication and authorisation, particularly in scenarios where user delegation is required. It allows applications to access user data without exposing credentials, supports fine-grained permission scopes, and integrates well with web-based authentication flows. However, OAuth 2.0 introduces additional complexity, requiring a dedicated authorisation server, token management, and an interactive login process, which may not always align with an API's intended use cases [21].

API keys, on the other hand, provide a simpler authentication mechanism. They do not require interactive authentication and are well-suited for server-to-server communication, internal integrations, and machine-generated requests. However, they lack built-in mechanisms for fine-grained access control and require additional security measures to prevent unauthorised use [21].

While most resources for REST API design and security recommend OAuth2.0 [38, 41, 70], it will not be implemented for the Albion Online Public API in the scope of this thesis. None of the use cases laid out by the requirements in the previous chapter would utilise OAuth's benefits over API keys. OAuth's primary strength lies in allowing users to grant permissions to third-party applications without sharing credentials, which is unnecessary in this context. Additionally, OAuth workflows introduce additional

implementation overhead, requiring a centralised identity provider and token exchange mechanisms that are also not essential for the targeted use cases of this project [18, 21]. As stated in subsection 2.2.2, the operational and development costs are opposing factors for the implementation of a public game API. With the complexity that OAuth entails, the development costs are expected to be higher than for API keys.

Given the nature of the Albion Online Public API, API keys present a more practical and efficient authentication method. API keys allow third-party developers to integrate the API with minimal friction, facilitating access without requiring interactive authentication flows [21]. This also enables integrations with tools like Google Sheets that are restricted in their ability to do complicated authentication handshakes but are still commonly used for data processing[1]. In comparison to OAuth, the simplicity of API keys also provides a higher level of maintainability for internal developers and, therefore, a higher business value.

In addition to this, it should be stated that a combination of the two mechanisms may be implemented in future iterations should other requirements arise. When doing so, the following statement should be taken into consideration:

> API keys are very convenient for devs. Sadly, EVE lost several notable long term projects and many smaller ones when it finally shut down the XML API and forced the move to OAuth and ESI. To give very broad numbers on that lossage: 25% didn't want to sign a developer agreement and 25% were doing things that the new api deliberately didn't expose the capabilities for. Those groups were never going to survive the migration. However, 50% of the lossage was just that people have limited time and it sucks to work on something for years only for the API vendor to change fundamental things on you. People don't always have time in their life for doing huge rewrites, especially if they've moved on from the game and are mostly maintaining the project out of fondness for the community.
>
> (Timothy Suthers, personal communication, February 2025)

Ultimately, while OAuth 2.0 remains an industry standard for user authentication and delegated authorisation, the specific requirements of the Albion Online Public API align more closely with the simplicity and efficiency of API keys. Therefore, the design intends the usage of API keys. To not interfere with other authentication methods already implemented in the API Gateway, the key should be included in a custom header called `x-albion-api-key`. Additionally, the API keys should have an expiry date that users are required to set on API key creation.

As required per FR-2, API Key scopes will be provided through a custom implementation in the Auth Service. This will only become relevant once a second use case from FR-1 is implemented.

[1] see for example this Forum Post

6.2.2 Rate Limiting

As stated in FR-3, the Albion Online API must apply rate limiting to its endpoints. This is done via the API Gateway. For now, it was decided that all clients should receive the same limits.

The technical design does not demand any concrete rate limits but rather requests that all relevant numbers be configurable to be able to adjust quickly. Should the load on the new API be lower or higher over time, or should the performance vary from the expected state, developers should be able to adjust the rate limits without having to change any code.

Additionally, the design states that rate limits should apply per API key and per IP address. This is due to the circumstance that the API Gateway will not only deliver the API endpoints but also the API documentation as OpenAPI specification and Swagger UI, which needs to be available for anonymous users. Applying rate limits per IP address prevents anonymous users from maliciously or accidentally Denial of Service (DOS)-attacking the gateway via the documentation.

With this, the rate limits should apply to all endpoints exposed through the API Gateway.

6.2.3 Versioning

While versioning could be addressed when the first breaking change is made, it is best practice to establish a version identification strategy from the start (see subsection 3.1.3). For this project, URI Versioning was selected as a strategy. While it does not strictly comply with REST principles [69, p. 205], it is very simple to implement and use. Additionally, this approach is already in use for internal APIs in the web infrastructure at Sandbox Interactive.

6.2.4 Endpoint Design

The endpoint design was done by analysing the current kills endpoints in the GIS, discussing with game designers, and adjusting according to their input and best practices determined in section 3.1. The GIS currently offers six endpoints under the `/events` path. These are the endpoints delivering kill information. The naming suggests that other events can be requested through these endpoints as well, but that is not the case. Table 6.1 provides an overview of the `/events` endpoints, which are all available through the HTTP GET method.

URI	Parameters	Description
`/events`	**limit:** 1 - 51 **offset:** 1 - 1000 **guildId:** Restrict to a guild	Recent PvP kills
`/events/{ID}`	None	Information about a specific event
`/events/killfame`	**range:** week | month | lastWeek | lastMonth **limit:** 1 - 51 **offset:** 1 - 1000	Top PvP kills within the selected time range
`/events/playerfame`	**range:** week | month | lastWeek | lastMonth **limit:** 1 - 51 **offset:** 1 - 1000	Top players within the selected time range based on kill fame
`/events/guildfame`	**range:** week | month | lastWeek | lastMonth **limit:** 1 - 51 **offset:** 1 - 1000	Top guilds within the selected time range based on kill fame
`/events/playerweaponfame`	**range:** week | month | lastWeek | lastMonth **weaponCategory:** all | <weaponCategories ID> **limit:** 1 - 51 **offset:** 1 - 1000	Top players with most kill fame within the selected time range and weapon category

Table 6.1: *Kills endpoints in the Game Info Service, as documented by [24]*

Conversely to what the URIs suggest, `/events/playerfame`, `/events/guildfame` and `/events/playerweaponfame` do not return kill data, therefore they can be left out of the discussion for now.

The `/events/killfame` endpoint does not follow the REST API design best practice to apply sorting and filtering through query parameters. Instead, a path element is used to select the sorting method for the events. This mechanism should not be replicated in the new API, which should rather allow the selection of sorting method in query params in a `/kills` base endpoint.

Listing A.1 shows a sample kill event returned by the GIS. After discussing with the Game Design team at Sandbox Interactive and reviewing best practices, the following

changes will be applied to the shown schema:

- All keys should use `camelCase`, rather than a mixture of `camelCase` and `PascalCase`.
- `groupMemberCount` and `numberOfParticipants` will be removed since they can easily be reconstructed by counting the elements in `Participants` and `GroupMembers`
- `EventId` will be renamed to `id`, to reinforce its use as identifier for kills.
- `Version`, `BattleId`, `Category` and `Type` will be removed due to irrelevancy in the new API.
- the `Location` is currently always null to prevent abuse. This will be changed, and instead, the name of the cluster in which the kill happened will be provided.
- The `DeathFame`, `KillFame` and `FameRatio` fields in the `Killer` and `Victim` objects will be removed since they can be calculated from other provided data.
- The `LifetimeStatistics` in the player objects will also be removed as they do not add any relevant information to a kill.

Depending on the current load on the GIS, the endpoints return kills older than five minutes. According to the game designers at Sandbox Interactive, this behaviour should remain for the new API, and will additionally be implemented as a configurable parameter, should the need arise to change this convention. This delay prevents players from gaining unfair advantages through the use of third-party tools.

> We share combat results to give players reliable information, but we also delay them by a few minutes and obscure the exact location to protect the integrity of open-world PvP. This way, third parties, who were not part of the initial fight, can not abuse the information to easily track down the survivors, who are likely weakened and/or weighed down by their loot.
>
> (Moritz Bokelmann, Game Director at Sandbox Interactive, March 2025)

Additionally, the new `/kills` endpoint should return less information on each kill than when requested through `/kills/{ID}`. Only the data that can easily be shown in a list format or that is required to request further information on the kills, or involved players should be included.

The following paragraphs present the resulting design of the kills endpoints in full detail.

Retrieving a List of Kills

The `GET /datacenter/v1/kills` endpoint returns a paginated list of recorded kills. Clients can apply filters based on time range and sorting preferences. Pagination parameters control the number of results returned per request.

```
GET api.albiononline.com/datacenter/v1/kills?time-range=today&sort=total-
    victim-kill-fame-desc&page=0&size=25
```

Listing 6.1: *Example Request to Retrieve Kills*

Query Parameters:

- **`time-range`** (optional): Specifies the time range for kills. Allowed values: `today`, `yesterday`, `this-week`, `last-week`, `this-month`, `last-month`
- **`sort`** (optional): Defines sorting order. Allowed values: `timestamp-desc` (default) and `total-victim-kill-fame-desc`.
- **`page`** (optional): The page number (default: 0).
- **`size`** (optional): The number of results per page (default: 25, max: 50).

A successful request returns a JSON object containing a paginated list of kills.

```
{
  "content": [
    {
      "id": 0,
      "killTimestamp": "2025-03-11T13:44:55.774Z",
      "killer": {
        // reduced player data
      },
      "victim": {
        // reduced player data
      },
      "location": "string",
      "totalVictimKillFame": 5000,
      "killArea": "string"
    }
  ],
  "size": 25,
  "page": 0,
  "hasNext": false,
  "hasPrevious": false
}
```

Listing 6.2: *Example Response for Kills List*

```
{
  "averageItemPower": 200,
  "name": "string",
  "characterId": "3fa85f64-5717-4562-b3fc-2c963f66afa6",
  "guildName": "string",
  "guildId": "3fa85f64-5717-4562-b3fc-2c963f66afa6",
  "allianceName": "string",
  "allianceId": "3fa85f64-5717-4562-b3fc-2c963f66afa6",
  "avatar": "string",
```

```
  "avatarRing": "string",
}
```

Listing 6.3: *Example Reduced Player Data*

Retrieving a Single Kill Record

The `GET /datacenter/v1/kills/{id}` endpoint fetches detailed information about a specific kill event. The request requires a valid kill ID.

```
GET /datacenter/v1/kills/12345678
```

Listing 6.4: *Example Request for a Single Kill*

A successful response returns detailed data on the kill, including participants, equipment used, and kill Fame.

```
{
  "id": 0,
  "killTimestamp": "2025-03-11T13:31:11.235Z",
  "killer": {
    // player data
  },
  "victim": {
    // player data
    "inventory": [
      {
        "itemType": "string",
        "count": 1,
        "quality": 1
      }
    ]
  },
  "location": "string",
  "totalVictimKillFame": 0,
  "killArea": "string",
  "participants": [
    {
      // player data
      "damageDone": 100,
      "supportHealingDone": 0
    }
  ],
  "groupMembers": [
    {
      // player data
    }
  ]
}
```

Listing 6.5: *Example Response for a Single Kill*

```
{
  // reduced player data
  "equipment": [
    {
      "item": {
        "itemType": "string",
        "count": 1,
        "quality": 1
      },
      "equipmentSlot": "string"
    }
  ]
}
```

Listing 6.6: *Example Player Data*

These endpoints are compliant with REST principles, best practices and industry standards discussed in section 3.1.

6.3 API Key Management

Since the API design provides for the use of API keys, a user interface must be implemented in which users can manage their keys. This interface should include the ability to list, create and delete API keys, as well as copy them to the clipboard. The interface should be integrated into the Albion Online website, more specifically into the account management section. The page should link to the official API documentation, and inform users about the ToS.

Figure 6.3, Figure 6.4, Figure 6.5 and Figure 6.6 show mockups of the interface to be implemented.

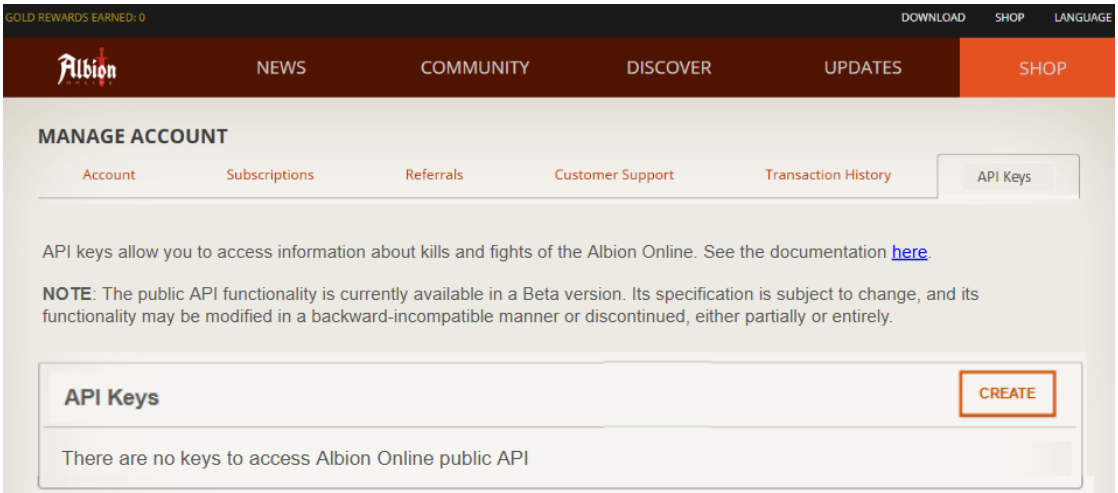


Figure 6.3: *Mockup 1 of the API key management page*

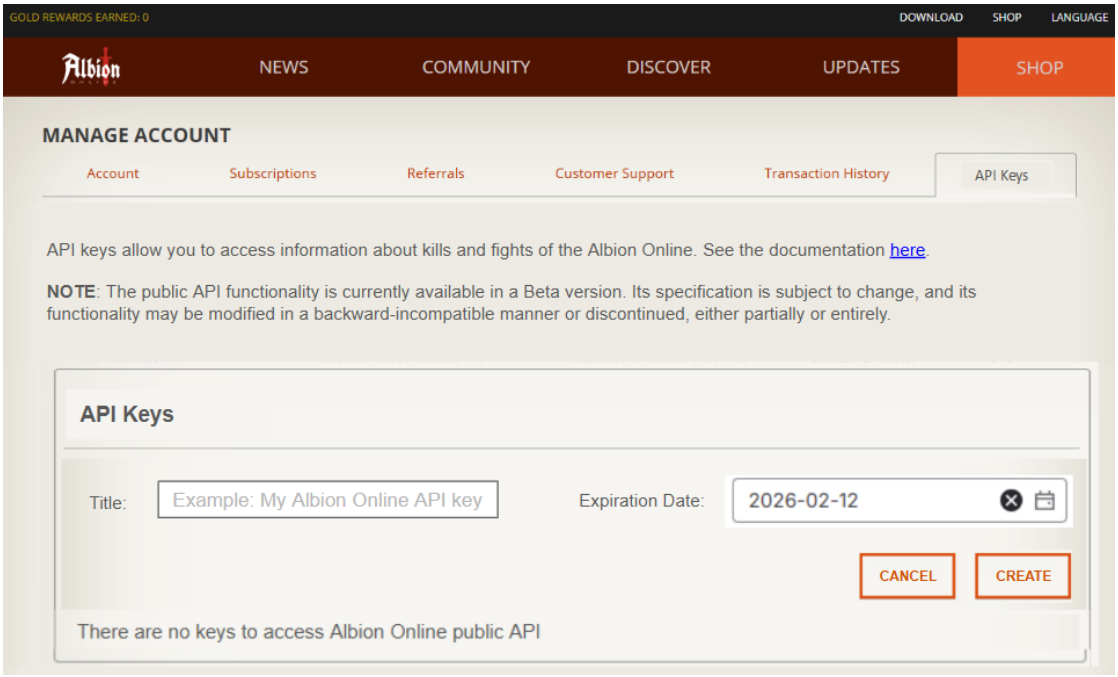


Figure 6.4: *Mockup 2 of the API key management page*

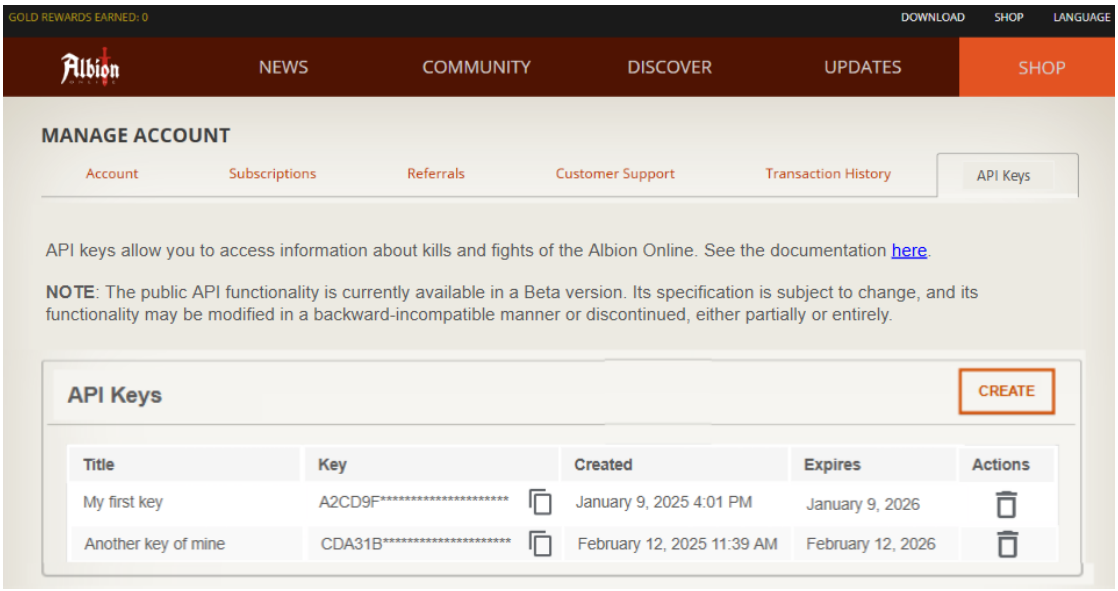


Figure 6.5: *Mockup 3 of the API key management page*

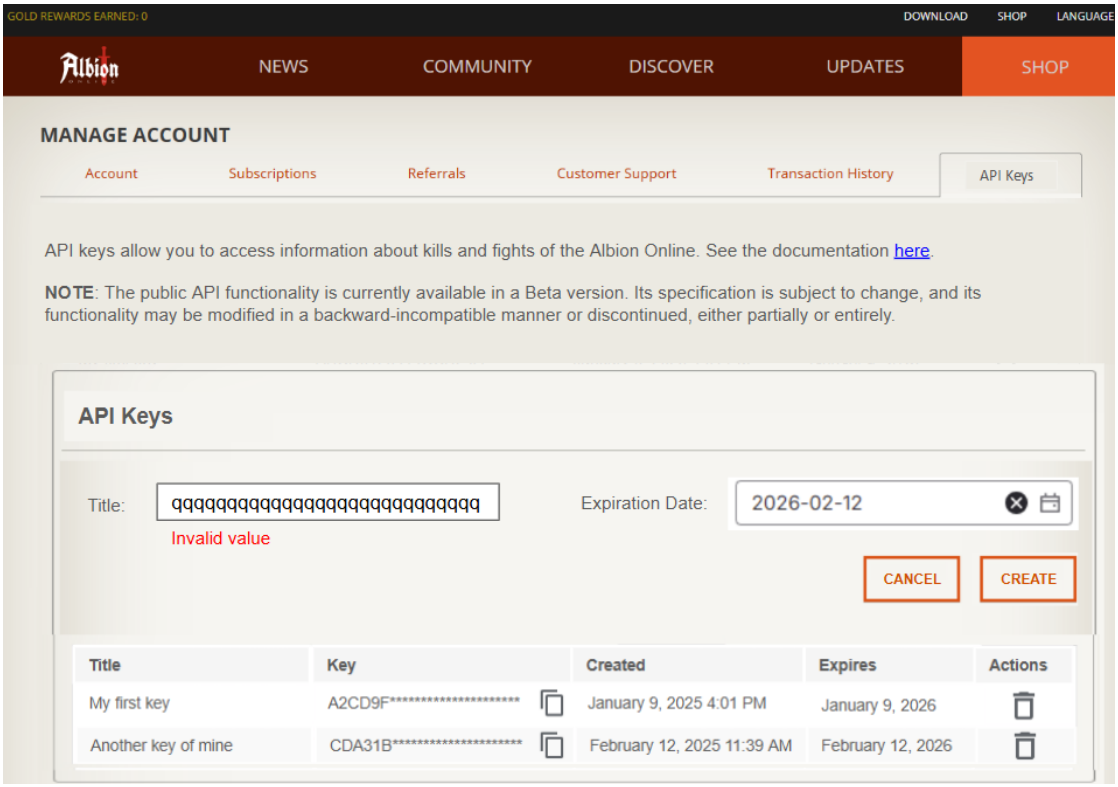


Figure 6.6: *Mockup 4 of the API key management page*

In the future, customer support agents at Sandbox Interactive should also be enabled to see and manage users API keys. This feature should be added in the CS Tool. Furthermore, to comply with C-1 and thus General Data Protection Regulation (GDPR), API keys have to be deleted when a user requests an account deletion.

6.4 Technologies and Frameworks

As defined by C-3 and C-4, the technologies and frameworks for this project are selected to integrate well with the already existing infrastructure at Sandbox Interactive. This ensures compatibility with the current ecosystem, reducing operational overhead and improving maintainability.

Backend Framework

The *Stats-ETL-Service*, *Stats-Read-Service*, and *Auth Service* are implemented using **Spring Boot 2** with **Java 8**. Both are used to maintain compatibility with the existing codebase, despite newer Java versions being available. While Java 17 offers enhanced performance and language features, upgrading would require significant refactoring of existing systems, which cannot be done in the scope of this thesis. The web team at Sandbox Interactive has written many Spring Boot 2/Java 8 compatible tools and libraries over

the years, that can be utilised here.

Database and Storage Technologies

As a storage solution, all new databases will rely on PostgreSQL. This is in alignment with the current web infrastructure but is also intended to provide an advantage over the Elasticsearch solution implemented in the GIS (see subsection 4.1.4).

ETL and Data Processing

The *Stats-ETL-Service* is responsible for extracting, transforming, and loading game statistics, and should do so in batches. The Spring Batch library promises to provide corresponding functionality, and should therefore be employed.

API Gateway

The *API Gateway* is partly already implemented using **Spring Cloud Gateway**. Since there is currently no reason provided to switch to a different solution, this shall remain the same.

Deployment

All web services at Sandbox Interactive are built and then deployed with Jenkins and server/service configuration is handled via Puppet. These tools will be utilised for the new infrastructure as well.

Monitoring and Logging

Monitoring and logging are integrated using the company's ELK stack, and Datadog.

This technology stack provides a robust and maintainable foundation for the public API.

6.5 Database Schema

The schema of the new `albion_stats` database is modelled after the required data for the API endpoints. Fields that allow sorting or filtering in the API are indexed so that querying the database is faster. This currently applies to the `killTimestamp` and `totalVictimKillFame`. Figure 6.7 depicts the schema to store the required kill data.

At the core of the schema is the `kills` table, which records individual kill events. The `players` table stores player metadata, such as character name, guild affiliation, and total item power, while the `items` table maintains information on in-game equipment, including item type, quality, and enchantments. The `locations` table stores the exact in-game location at which the kill happened.

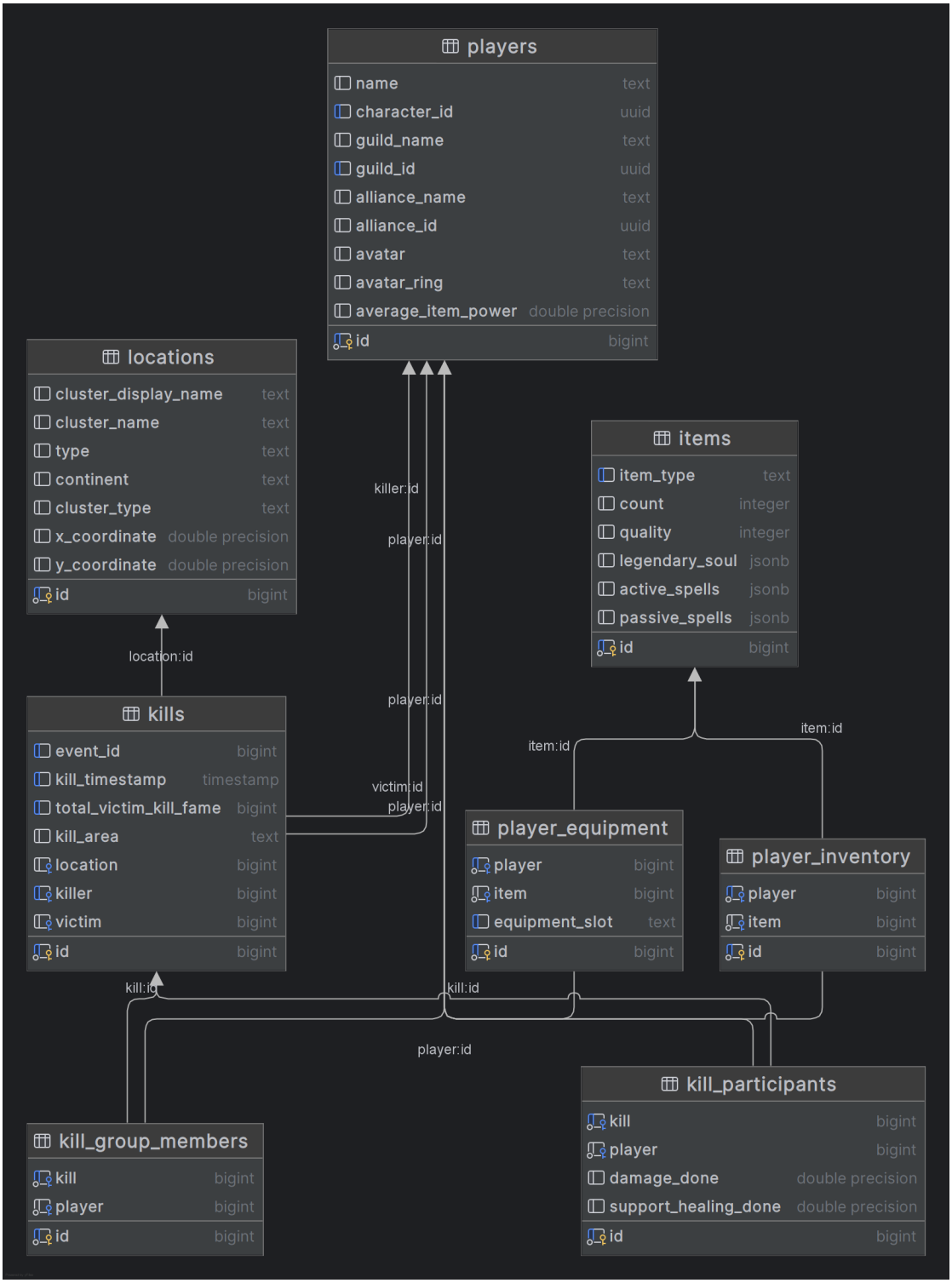


Figure 6.7: *Schema in the* `albion_stats` *database, exported from JetBrains DataGrip*

These tables are connected through various relationships. The `kill_participants` table associates all players who contributed to the death of the victim with a `kill`, storing damage and healing contributions. Similarly, the `kill_group_members` table

tracks the victim's group members. The schema also includes dedicated tables for player inventories (`player_inventory`) and equipped gear (`player_equipment`), both of which reference the `items` table.

6.6 Data Processing

The Stats-ETL-Service will be responsible for preparing the required data for the API. In the `dataexport` database, events currently have the format shown in Figure 6.8. Kill events can be queried through `data_type='KILL'`. The `data` field contains all relevant information in a JSON format. The structure of the JSON is currently undocumented and needs to be researched in the game server code, which is responsible for writing events to the `dataexport`.

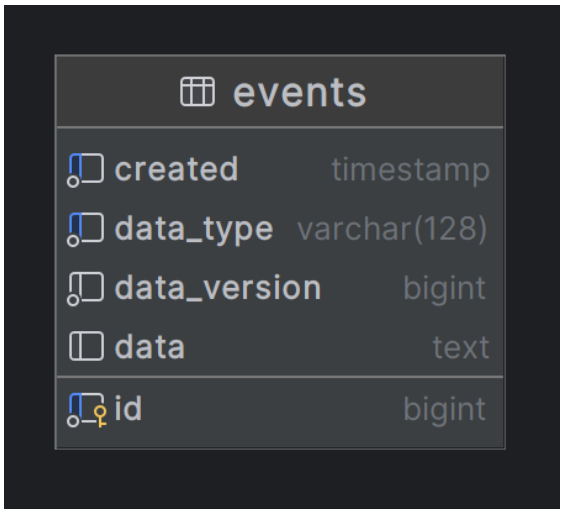


Figure 6.8: *Schema of the `events` table in the `dataexport` database, exported from JetBrains DataGrip*

This is the process the Stats-ETL-Service should therefore implement:

1. Request a kill event batch from the `dataexport` database
2. Deserialise JSON from the `data` field
3. Transform the data to the desired schema defined in Figure 6.7
4. Write the kill event batch to the `albion_stats` database
5. Repeat

The implementation should include configuration parameters for the batch size of this process and for the delay between batch requests to be able to adjust the produced load on the `dataexport` database as needed.

Additionally, a metric should be added to track the amount of kills processed, to provide developers with information on the health of the service, as well as the performance.

The architectural and design choices outlined in this chapter lay the foundation for a robust and efficient Albion Online Public API, addressing the challenges identified in earlier chapters. By leveraging established API design principles, implementing security measures, and optimizing data processing workflows, the proposed solution ensures that third-party developers and community members can access in-game data in a structured and reliable manner. Additionally, the modularity of the design allows for future extensions and adaptations as new requirements emerge or different use cases are implemented. The subsequent chapter will describe the implementation of this design, detailing the technical steps taken to realise the proposed architecture while ensuring adherence to the outlined requirements and best practices.

7 Implementation

In reality, design and implementation were parallel processes, despite what the structure of this thesis suggests. Development was done as an iterative process, producing the idea for an approach, implementing, testing, and finding a new approach if necessary. The system architecture design (section 6.1) is a result of testing multiple ideas for the included components, and selecting solutions accordingly.

This chapter, therefore, only provides an overview of the solutions that were deemed the most important, leaving out various iteration steps in favour of showing the final implementation state of each component. This is necessary due to the high amount of work that was done for the project. Most of the omitted work can be recreated by following the best practices outlined in chapter 3, the requirements (chapter 5) and the design (chapter 6).

7.1 Service Setup

Since the web service template and the standardised process for setting up new web services at Sandbox Interactive was outdated, the implementation of the Public API infrastructure was blocked. To fix that, a new service template was created and pushed to a GitLab repository. The template is based on the structure and common features of already deployed services (User Service, Mail Service, etc.). It includes the following functionalities:

- Sample README.md file
- A `.gitignore` file
- Required Gradle files
- The established web service structure with `client`, `common` and `server` modules
- Base setup for integration testing in the `server` module
- Base security and API documentation configurations
- Base `application.yml`

Once the template was updated, the documentation for it was updated as well. This enabled and simplified the creation of all required Java services for this project.

7.2 Authentication Service

As defined in section 6.1, an Authentication Service (or Auth Service) is required to handle API keys. The service was set up with the help of the template as described in the previous section. Additionally, a corresponding `auth_service` database was created. The Auth Service mainly provides an internal CRUD API for API keys stored in that database. The CRUD operations can be used for the key management page on the Albion Online website. The structure of an API key is shown in Figure 7.1.

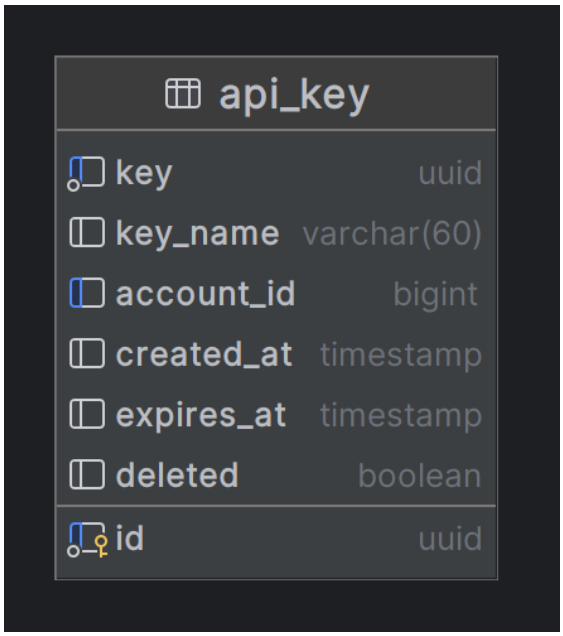


Figure 7.1: *Schema of an API Key, exported from JetBrains DataGrip*

The deletion of API keys is implemented as a "soft delete". This means that deleting an API key does not remove it from the database, but rather marks it as deleted in the corresponding database column. This prevents the unlikely but not entirely impossible case, that a Universally Unique Identifier (UUID) that was used before is handed out to a different user.

The Auth Service also incorporates logic for key validation, which is used by the API Gateway. The following criteria need to be met for an API key to be valid:

- The key exists in the database.
- The expiration date of the key is in the future.
- The deletion status is `false`.

7.3 API Gateway

Similarly to other Spring Boot Services at Sandbox Interactive, the API Gateway repository is split into `client`, `common` and `server` modules. The main difference is that the

`server` module utilises the `spring-cloud-starter-gateway` dependency, which requires the service to run on Spring Webflux and, therefore, a reactive web stack. The Spring Cloud Gateway provides "a simple, yet effective way to route to APIs and [...] cross cutting concerns to them such as: security, monitoring/metrics, and resiliency" [13]. This section describes how these functionalities are configured for the Albion Online Public API.

7.3.1 Dynamic Route Registration

The common style of registering routes in Spring Cloud Gateway is via configuration in the `application.yml`. That means whenever a new route is supposed to be registered, the API Gateway needs to be re-configured and re-deployed. While that would not cost significant effort, it does imply a form of coupling between the gateway and the proxied services. To eliminate that coupling, a Dynamic Route Registration system was implemented.

The system is designed to include two main features:

1. The API Gateway is able to receive route information via REST API, and subsequently proxies those routes. This was already implemented in a basic form prior to this project in the API gateway `server` module.
2. An endpoint that is supposed to be proxied by the gateway can automatically be registered via an annotation. To enable services to use this functionality, this feature needs to be implemented in the API gateway `client` module.

The following paragraphs outline the steps taken to implement the second feature and the modifications made to the implementation of the first feature.

First, the annotation interface `RegisterAtGateway` was added to the gateway client, as shown in Listing 7.1. With this, the RetentionPolicy allows the annotation to be available to read at runtime. The annotation can be used to register routes of singular controller methods as well as full controller classes.

```
1 @Retention(RetentionPolicy.RUNTIME)
2 @Target({ElementType.METHOD, ElementType.TYPE})
3 public @interface RegisterAtGateway { }
```

Listing 7.1: *RegisterAtGateway annotation interface*

Next, an `ApplicationReadyEvent` listener was set up that collects all endpoints annotated with `RegisterAtGateway` and sends those to the gateway. Additionally, the OpenAPI documentation route is fetched and sent to the gateway, so that it can be proxied as well. Listing 7.2 shows a pseudocode version of the implementation for brevity. In reality, an abstract `BaseRegisterRoutesOnStartup` class was written, that is implemented by a class for Spring MVC and Spring WebFlux each. That way, services that utilise either or both of these libraries can register routes.

```
public class RegisterRoutesOnStartup implements ApplicationListener<
    ApplicationReadyEvent> {

    @Autowired
    private RequestMappingHandlerMapping handlerMapping;

    @Value("${server.protocol:http}://${server.address}:${server.port}")
    protected String applicationBaseUrl;

    @Override
    public void onApplicationEvent(ApplicationReadyEvent event) {
        Map handlerMethods = handlerMapping.getHandlerMethods();
        List<RouteSpecificationDto> routes = new ArrayList<>();
        for (Map.Entry entry : handlerMethods.entrySet()) {
            if (!isClassAnnotated(entry, RegisterAtGateway.class)
                && !isMethodAnnotated(entry, RegisterAtGateway.class)) {
                continue;
            }
            RequestMappingInfo requestInfo = entry.getKey();
            for (PathPattern pathPattern : requestInfo.getPathPatterns()) {
                for (RequestMethod method : requestInfo.getMethods()) {
                    routes.add(new Route(
                        pathPattern,
                        method,
                        applicationBaseUrl,
                    ));
                }
            }
        }
        // add api doc route
        // send all routes to API Gateway route registration endpoint
    }
}
```

Listing 7.2: *Pseudocode RegisterRoutesOnStartup class*

Lastly, this class needed to be registered as a Spring Bean in the service that utilises the API gateway client, so that the previously shown code would actually be executed on service startup. This was done via an `AutoConfiguration` class.

With this, a Spring service that wants to register its endpoints at the API Gateway, needs to follow this procedure:

1. Add the `api-gateway-client` as a dependency
2. Configure the required properties, as shown in Listing 7.3
3. Annotate the desired controller methods or classes with `@RegisterAtGateway`

```
albion:
  api-gateway:
    client:
```

```
    url: http://api.sample.com
    username: apiGatewayAdmin
    password: apiGatewayPassword
```

Listing 7.3: *Configuration of the API Gateway client*

This system allowed completely independent implementation processes of the API Gateway and the Statistics-Read-Service.

7.3.2 Security

Security is a critical component of the API Gateway and ensures that only authorised clients can access protected resources. The API Gateway enforces authentication and authorisation through a combination of API key validation, Basic Authentication and OAuth2.0.

The Basic Authentication and OAuth configuration were already present prior to this project. The former authentication method is provided by the service template and enables internal service communication. The latter is a undocumented custom OAuth implementation that enables customer support agents to authenticate using their CS Tool credentials and retrieve player-related information. This is already in use in a plugin for Zendesk in the live infrastructure, which is why changes to the security configuration need to be tested thoroughly.

Alongside the previous configuration, as defined in the design, API key authentication is implemented. This allows third-party developers to authenticate their applications via a static key included in a custom request header. The API key validation is implemented as a filter that intercepts incoming requests, ensuring that only clients with valid credentials can interact with protected endpoints.

In addition to OAuth2.0 and API Key authentication, the API Gateway also provides the internal Basic Auth strategy included in all web services at Sandbox Interactive. This needs to remain a possibility so that the Dynamic Route Registration endpoints can be exposed to internal infrastructure.

With these three approaches included in a single project, the resulting `SecurityConfig` class in the API Gateway is quite complex. It has been included in Listing A.2.

This configuration ensures that:

- Administrative endpoints (e.g., `/v*/admin/**`) are restricted to users with the `ADMIN` role.
- Endpoints for customer support (e.g., `/v*/*/forZendesk`) require authentication via OAuth 2.0.
- Public endpoints such as the Swagger interface and OpenAPI specifications remain accessible to unauthenticated users.
- API key authentication is required for all other endpoints.

7.3.3 Rate Limiting

The rate limiting in the API Gateway is set up hand in hand with the route registration and the security configuration. Routes can register with weights, so that higher weighted routes have lower rate limits, and rate limiting is applied per IP and API key, as defined in the design.

Basic rate limiting was already implemented before this project, using the included functionality of the Spring Cloud Gateway library. As per documentation, the library "uses a `RateLimiter` implementation to determine if the current request is allowed to proceed. If it is not, a status of `HTTP 429 - Too Many Requests` (by default) is returned. This filter takes an optional `keyResolver` parameter and parameters specific to the rate limiter [...]. `keyResolver` is a bean that implements the `KeyResolver` interface. [...] The `KeyResolver` interface lets pluggable strategies derive the key for limiting requests" [13]. The `KeyResolver` implementation for the API Gateway is shown in Listing A.1.

7.4 ETL Process

The implementation process for the Stats-ETL-Service was not straightforward. As was already noted in section 6.1, the system architecture is designed in a way that might cause performance issues through latency in the ETL process. This section therefore outlines all stages and steps that were taken to prepare the required data in an acceptable time frame (see NFR-10).

The Stats-ETL-Service is setup as per section 7.1, alongside the `albion_stats` database. Since the Stats-ETL-Service owns the new database, the schema designed in section 6.5 is applied via Liquibase migrations. The full SQL required for the database setup is provided in Listing A.4.

As can be seen in the system architecture design (Figure 6.2), the Stats-ETL-Service has to communicate with two databases: The `dataexport` database to read from, and the newly configured `albion_stats` database to write in. This dual datasource setup requires additional configuration in Spring Boot. A tutorial by Kevin Wittek on Baeldung [82] was used to implement a solution, where the connection to `albion_stats` is set as the primary data source. Entities for kill events in the `dataexport` database and all relevant tables in the `albion_stats` database were created.

With the connection to the `dataexport` database established, the question about the structure of the kill data needed to be answered (see section 6.6). For this, the code of the game server was pulled and evaluated. The schema was then implemented in POJOs, and the Jackson library used to deserialise the JSON.

From there, the data needed to be transformed into the target entities. For this, the MapStruct library was used. The mapper code is appended in Listing A.2. With that, the "transform" stage of the ETL process is completed. The corresponding method in

the `KillMappingService` class is provided in Listing 7.4.

```
1     private Kill fromKillEvent(KillEvent killEvent) {
2         try {
3             KillEventData killEventData = objectMapper.readValue(
4                 killEvent.getData(),
5                 KillEventData.class
6             );
7             Kill kill = KillMapper.INSTANCE.fromEventData(killEventData);
8             kill.setEventId(killEvent.getId());
9             return kill;
10        } catch (JsonProcessingException e) {
11            log.error("KILL_EVENT_DATA_PARSE_ERROR", e);
12        }
13    }
```

Listing 7.4: *Kill transformation method in the KillMappingService class*

This was first tested through an API endpoint in the Stats-ETL-Service to extract, transform and load a single kill from the `dataexport` database. After confirming that the loaded data was complete and correct, the batch processing was implemented.

7.4.1 Spring Batch Configuration

Initially, the Stats-ETL-Service utilised the Spring Batch library. "Spring Batch provides reusable functions that are essential in processing large volumes of records, including logging/tracing, transaction management, job processing statistics, job restart, skip, and resource management. It also provides more advanced technical services and features that will enable extremely high-volume and high-performance batch jobs through optimisation and partitioning techniques. Simple as well as complex, high-volume batch jobs can leverage the framework in a highly scalable manner to process significant volumes of information" [13]. These functionalities are a good fit for the design established in section 6.6.

The required configuration was set up (see Listing A.3) to read and write the data from and to the desired databases. Additionally, code was written to establish restart safety in batch processing. That way, if a new version was deployed or if the service ever crashed, it would be ensured that no duplicate kills would be processed and written to the `albion_stats` database.

Additionally, a Datadog metric was added to track the number of kills that are being processed.

7.4.2 Performance Improvements

After deploying this initial functioning implementation of the ETL process in the staging environment in US, it was noticed that the performance was worse than expected. The

metric that was added showed a throughput of 8,000 kills per minute, which means it would have taken around 23 days to process all kills in the Live US `dataexport` database. This was not acceptable in regards to NFR-10. Therefore, a few changes were made and tested to improve the performance.

First, Spring Batch was eliminated. While it is intended to be used for high-performance batch processing, it proved to be a bottleneck in this situation. Due to the tracking of the batch tasks in the `albion_stats` database, a lot of read and write queries were added on top of an otherwise simple process. This overhead would have had even more impact in the Amsterdam and Singapore data centers due to the latency between the Stats-ETL-Services there and the `albion_stats` database in the US. By removing Spring Batch, a lot of overhead code execution could be eliminated.

It was also noticed that inserts in the `albion_stats` database were done row-by-row, which costs a lot more time than if the inserts were batched. According to [3], Java Database Connectivity (JDBC) can enable batch inserts with Hibernate and JPA through the following steps:

1. Set the `hibernate.jdbc.batch_size` property in the data source configuration to the desired batch size
2. Switch the identifier generator of the entities. Using `GenerationType.IDENTITY` as generation strategy disables batch inserts.

Therefore, `GenerationType.SEQUENCE` was implemented for all `albion_stats` entities. A Liquibase migration was written to alter the increment values of the relevant sequences, as shown in Listing 7.5, and a `SequenceGenerator` with the same `allocationSize` was added to the identifiers in each entity. An example of this is shown in Listing 7.6.

```
--liquibase formatted sql

--changeset pauline.roehr:20250205_1516_AO-40405_set_sequence_increments.sql
    failOnError:true runInTransaction:true

alter sequence kills_id_seq increment by 100;
alter sequence items_id_seq increment by 100;
alter sequence kill_group_members_id_seq increment by 100;
alter sequence kill_participants_id_seq increment by 100;
alter sequence locations_id_seq increment by 100;
alter sequence players_id_seq increment by 100;
alter sequence player_equipment_id_seq increment by 100;
alter sequence player_inventory_id_seq increment by 100;
```

Listing 7.5: *SQL Migration to alter sequence increments*

```
    @Id
    @GeneratedValue(
        strategy = GenerationType.SEQUENCE,
```

```
4         generator = "kills_id_generator"
5     )
6     @SequenceGenerator(
7         name="kills_id_generator",
8         sequenceName = "kills_id_seq",
9         allocationSize = 100
10    )
11    private Long id;
```

Listing 7.6: *Kill identifier field*

With these changes, the Stats-ETL-Service was re-deployed to the US staging environment, and the throughput was measured again. Thanks to the implementation of configurable parameters for the batch read size and the `hibernate.jdbc.batch_size` it was possible to simply test various configurations. The results of these tests are shown in Table 7.1.

Applied Configuration	**Result**
Read Size: 100 **Insert Batch Size:** 1,000	12k kills/minute
Read Size: 1,000 **Insert Batch Size:** 1,000	21k kills/minute
Read Size: 5,000 **Insert Batch Size:** 1,000	OutOfMemoryException
Read Size: 2,000 **Insert Batch Size:** 1,000	22k kills/minute
Read Size: 2,000 **Insert Batch Size:** 10,000	22k kills/minute

Table 7.1: *Throughput results of different configurations in the Stats-ETL-Service*

With the best result of an average of 22,000 kills per minute, the processing of all kills in the Live US environment would have taken around 8.5 days. This was already a big improvement but did not meet the requirement of a maximum of 5 days (NFR-10). Therefore two further improvements were made to the code:

1. Usage of `Slice` instead of `Page` in the `KillEventRepository` when requesting kill events from the `dataexport` database. `Pages` execute a count query on the requested table each time, which is not needed for the ETL process.
2. Apply multithreading for the scheduled tasks and for the Java streams that were used.

The resulting code is provided in Listing A.4. With this, an average throughput of 97,000 kills per minute was reached in the staging environment and, therefore, an estimated time of 2.5 days to process all Live US kills.

7.4.3 Latency Issues

When the servers for the Stats-ETL-Service in Amsterdam and Singapore arrived, and the service was deployed on each, more performance issues were noticed, as was already expected in section 6.1. Each of these servers processed an average of 2,000 kills per minute. On Singapore Live, this would have taken roughly 22 days to process 68 million kills, and on Amsterdam Live about 11 days for 33 million kills. Again, this did not comply with NFR-10.

To identify the exact bottleneck, JProfiler was used on the Stats-ETL-Service in Staging Singapore. As shown in Figure 7.2, the execution of `select nextval('xx_id_sequence')` was recognised to take an average of 213 milliseconds. It was concluded that the execution of this query is much slower than the inserts because a response to the query is expected synchronously, whereas inserts can be handled asynchronously. While the number of sequence calls was already reduced when switching to `GenerationType.SEQUENCE` with `allocationSize=100`, this still slowed down the processing significantly.

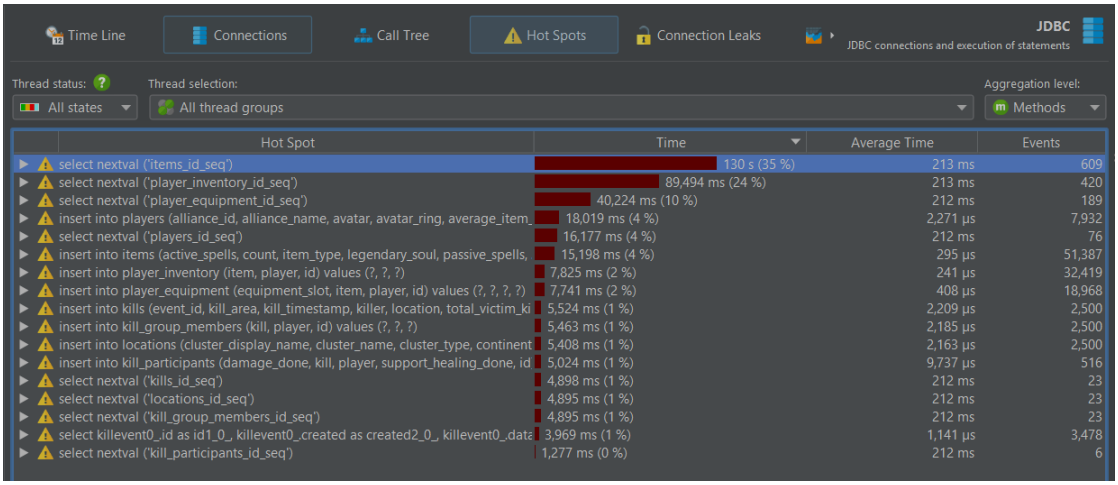


Figure 7.2: *JProfiler Hot Spots in JDBC Connections in the Stats-ETL-Service*

A first attempt to increase the allocationSize and sequence increments to 1000 resulted in a small throughput improvement to 2,500 kills per minute. Therefore, different measures had to be taken. The following ideas were discussed and attempted to implement in the web team:

- Write a custom solution without Hibernate, using basic JDBC, requesting only as many IDs as necessary per batch.

- Utilise a stored procedure in the `albion_stats` database
- Disable sequence generation on the side of the PostgreSQL database and generate IDs in the Java application. A requirement for this is that the Stats-ETL-Service is the only one writing to the corresponding tables. This is true according to the system architecture, but there is still a risk of developers inserting rows into the database by hand.

None of these ideas could be implemented within the time frame of this project.

Instead, an entirely different approach was taken that fulfilled the time constraint set by NFR-10. Databases in Singapore and Amsterdam were set up, to which the Stats-ETL-Services each were configured to write. Figure 7.3 shows the kill processing counter per environment, with data points for every 10 minutes. The processing was started at 12 pm on February 26. The Stats-ETL-Services in Singapore and Amsterdam had to be stopped roughly an hour later because the servers were running out of disk space. Since these two servers are Cloud Virtual Machines (VMs), it was possible to quickly add more disk space and restart the ETL processes after 20 minutes. With this, Amsterdam kill processing was finished around 6 pm and Singapore around 11:30 pm on the same day. The kills in the US were fully processed around 7:40 am on February 28, after about 44 hours. Each of the services provided an average throughput of 100,000 kills per minute.

Figure 7.3: *Kill Processing in all three live environments; Exported from Datadog*

After that, the processed data in Singapore in Amsterdam still had to be moved

to the `albion_stats` database in the US. For this, `pg_dump` and `pg_restore` were executed on the server in the US, as shown in Listing 7.7.

```
pg_dump --host SINGAPORE_HOST --port 5432 --username stats_etl_service --
    format=custom albion_stats --schema=live02 --file=/home/pauline_roehr/
    live02.dump
pg_dump --host AMSTERDAM_HOST --port 5432 --username stats_etl_service --
    format=custom albion_stats --schema=live03 --file=/home/pauline_roehr/
    live03.dump

pg_restore --host localhost --port 5432 --username stats_etl_service --dbname
     albion_stats --schema=live02 -j 10 /home/pauline_roehr/live02.dump
pg_restore --host localhost --port 5432 --username stats_etl_service --dbname
     albion_stats --schema=live03 -j 10 /home/pauline_roehr/live03.dump
```

Listing 7.7: *pg_dump and pg_restore commands*

Once the data was imported, the Stats-ETL-Services in Singapore and Amsterdam were re-configured to write to the `albion_stats` database in the US and restarted. It took about 15 minutes for each service to catch up with the kills that happened since the `pg_dump` was completed.

Since then, all three services have continuously been processing kills as intended by the design. Through the workaround of dumping and restoring the data from Singapore and Amsterdam, it was still possible to comply with NFR-10.

7.5 REST API

The Stats-Read-Service is the last Java service to be implemented for this project and provides the required REST API for the project FR-1-MVP. The service is set up to fulfil the API design in section 6.2. The code for this mainly includes database queries via JPA and the RestController for the kills, as shown in Listing 7.8.

```
package com.albiononline.statsreadservice.controller;

// imports removed for brevity

@RegisterAtGateway
@AllArgsConstructor
@RestController
@RequestMapping("/${albion.stats-read-service.endpoint-prefix}/v1/kill")
public class KillController {

    private final KillService killService;

    @GetMapping("/{id}")
    public KillDto kill(@PathVariable long id) {
        return KillMapper.INSTANCE.toDto(killService.findByEventId(id));
    }
```

```
17
18      @GetMapping
19      public SliceDto<KillDto> getKills(
20          @RequestParam(name = "timeRange", defaultValue = "TODAY")
        KillTimeRange timeRange,
21          @RequestParam(name = "sort", defaultValue = "TIMESTAMP_DESC")
        KillSortOption sortOption,
22          @RequestParam(name = "page", defaultValue = "0") @Min(0) @Max(100)
        int page,
23          @RequestParam(name = "size", defaultValue = "25") @Min(1) @Max(50)
        int size
24      ) {
25          Slice<KillDto> kills = killService.findKills(
26              timeRange,
27              PageRequest.of(page, size, sortOption.getSort())
28          ).map(KillMapper.INSTANCE::toDto);
29          return new SliceDto<>(
30              kills.getContent(),
31              kills.getSize(),
32              kills.getNumber(),
33              page < 100 && kills.hasNext(),
34              kills.hasPrevious()
35          );
36      }
37  }
```

Listing 7.8: *KillController class in the Stats-Read-Service*

The Entities that were written for the `albion_stats` database in the Stats-ETL-Service were moved to its `common` library so that they can be reused here. Therefore no changes to the Stats-Read-Service are required should the schema of the database change.

The OpenAPI specification for these endpoints is currently auto-generated.

7.6 API Key Management

The website page where users can manage their API keys was not implemented by the author of this thesis but shall be introduced shortly nonetheless. The implementation adheres to the design introduced in section 6.3 and was realised in PHP with Twig templates to integrate into the website. It utilises the endpoints provided by the Auth Service for the provided CRUD operations. At the time of writing this, the page is available in all fifteen languages supported by Albion Online. Figure 7.4 shows the interface in English.

The interface is locked behind a configuration parameter so that the existence of the API is not prematurely exposed to users.

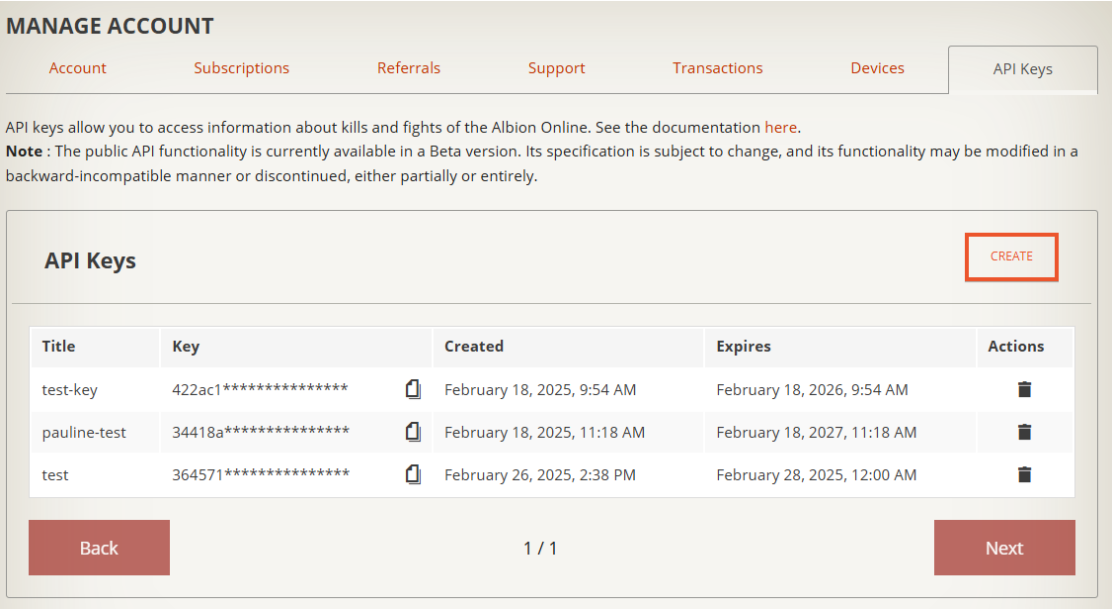


Figure 7.4: *API Key management interface on the website*

7.7 Deployment

As noted in section 4.3, builds and deployments are realised through Jenkins and Puppet in the Sandbox Interactive web team. After developing and testing locally and all required code review iterations, an initial deployment to staging was done with the following actions:

1. Request a new server in the US data center from GCore (done by a DevOps engineer)
2. Set up the server with base configuration (done by a DevOps engineer)
3. Add the required services and corresponding configuration to puppet
4. Roll out the puppet catalogue to the new server
5. Set up "Build" and "Deploy" Jenkins pipelines for each new service, configuring the latter to point to the new server
6. Build the required services with Jenkins
7. Deploy the required services with Jenkins

After testing this setup on staging US, the required infrastructure for staging in Amsterdam and Singapore was deployed as well. After several iterations of performance improvements as described in subsection 7.4.2, the deployments to the live environments were prepared. At the time of writing this, all configuration for a live Public API is prepared and is waiting for further testing before rollout.

8 Software Testing and Monitoring

At Sandbox Interactive, Quality Assurance (QA) teams for the game and for the web infrastructure usually have the biggest part in testing. Alongside the design and planning of a project, a test plan is set up and executed once a project nears its end. Despite the near-completion of the kills endpoints for the Albion Online Public API at the time of writing this, the test plan was not able to be executed within the time frame of this thesis. Therefore, this chapter will only provide a brief outline of the test plan that will be executed at some point in the future.

Alongside the testing done by professional software testers, software developers can contribute by doing code reviews or implementing unit and integration tests. Metrics, corresponding monitors, and informative logs also support the testing process. These techniques are all common practice for the web team at Sandbox Interactive. The concrete measures taken for this project will be documented in this chapter as well.

Additionally, this chapter includes a list of testing techniques that should be applied in the future.

8.1 Test Plan

The main goal of the test plan is to verify and validate the requirements for the project. Table 8.1 and Table 8.2 outline the most important test cases.

Requirement/Test Case	**Action**	**Expected Result**
FR-1-MVP: Data Access	Send an API request to retrieve kill data.	API returns an HTTP 200 OK and a kill data JSON response.
FR-1: Filtering	Request kill data using a query parameter for filtering	Response includes only the filtered subset of data matching the query.

Table 8.1: *Manual Test Plan for the Public API, Part 1*

Requirement/Test Case	Action	Expected Result
FR-1: Sorting	Request kill data using a query parameter for sorting.	Returned data is correctly sorted based on the provided sorting criteria.
FR-2: API Authentication	Attempt to access an endpoint without authentication.	Request is rejected with HTTP 401 Unauthorized.
FR-2: Role-Based Authorisation	Access an admin-only endpoint with a standard user account.	Request is rejected with HTTP 403 Forbidden.
FR-3: Rate Limiting	Send multiple requests exceeding the defined rate limit.	API responds with HTTP 429 Too Many Requests.
FR-7: Pagination	Request a list of kills using pagination parameters.	API returns the requested page with the provided amount of results.
FR-8: Error Handling	Send a request with invalid parameters.	API returns an error message with HTTP 400 Bad Request.
FR-9: Documentation Completeness	Visit the official API documentation	Documentation is available and provides guidance on how to use the API
FR-11: Secure Communication	Send a request to the API using HTTP.	The connection is refused and likely returns a `ERR_CONNECTION_REFUSED`
NFR-1: Response Time	Measure API response time for a standard request.	Response is returned within 1 second
NFR-6: Documentation Localisation	Visit the official API documentation	The documentation should be available in all desired languages
NFR-6: API Key Management Localisation	Visit the API key management page on the Albion Online website	The management page should be available in all desired languages
NFR-9: Data Consistency	Kill a player in the game and request that kill from the API	All returned kill data should be correct.

Table 8.2: *Manual Test Plan for the Public API, Part 2*

8.2 Developer Testing

While the execution of the test plan will be left in the hands of professional QA engineers, it is also possible to test various requirements as a software developer. This section lists the tests that were done in the scope of this master thesis. By no means are all requirements and edge cases covered with these, instead the applied measures ensure that not too many iterations have to be made before the systems can be rolled out to external users.

8.2.1 Manual Testing

NFR-1 was already tested in the local development and staging environments. On `localhost`, with a database of about 150,000 kills, requests to `/v1/kills` are responded to within 150 to 300 milliseconds, to `/v1/kills/{ID}` within 20 to 75 milliseconds. On staging, with the same size database, the response time ranges are 275-330ms and 130-160ms.

Additionally, NFR-9 was already partly tested during the implementation of the ETL process (see section 7.4). The processed data in the `albion_stats` database was deemed complete and correct when testing locally. In all environments, the database was also checked for duplicates through the following query:

```
SELECT event_id, COUNT(event_id)
FROM kills
GROUP BY event_id
HAVING COUNT(event_id) > 1;
```

Listing 8.1: *SQL query to check for duplicates in the kills table*

None of the environments returned any duplicate kill events.

A further testing measure that was implemented during the development process was the undertaking of code reviews. All code written within the scope of this thesis was reviewed by a web team member at Sandbox Interactive. While issues can always be missed during code reviews, they provide a layer of protection against bugs or other inconsistencies. Code Reviews also ensure stronger compliance with internal code conventions and best practices, therefore acting as a form of test for software quality requirements such as NFR-2, NFR-4, NFR-5, NFR-7, NFR-8 and NFR-11.

8.2.2 Unit and Integration Testing

As forms of automated testing, unit and integration tests were implemented in the Java services. All GitLab repositories are set up to run these tests whenever changes are made to the code, notifying the web team if a test fails. All Java applications also provide a basic integration test, that checks whether the Spring Boot application context can be loaded.

Service	Unit Tests	Integration Tests
API Gateway	Route Storage strategies (In-memory, JSON File), IP and API key resolution	Route Registration
Auth Service	API Key CRUD Methods (Controller and Service)	API key endpoints
Stats-ETL-Service	/	/
Stats-Read-Service	Calculation of kill time ranges	/

Table 8.3: *Automated Tests in the relevant Java services*

8.3 Monitoring and Logging

To ensure continuous system reliability and early detection of issues, monitoring and logging mechanisms have been integrated into the project. All Java services deployed within the system are equipped with availability monitoring through Datadog process tracking. These monitors verify whether the services are running as expected and send alerts to a Slack channel in case of a downtime that lasts longer than one minute. None of these monitors have been triggered since they have been set up.

Additionally, system resource usage is monitored, with alerts on low disk space availability and free inodes depletion across all servers. This reduces the risk of failures due to resource exhaustion. The newly deployed servers have also been integrated into the Web Overview dashboard within Datadog. This dashboard provides a centralised view of server health, request statistics, and potential performance bottlenecks, supporting system maintenance and troubleshooting.

A separate "Public API Metrics" notebook in Datadog was also created. Currently it provides information regarding kill processing in the Stats-ETL-Service, requests to and responses from `api.albiononline.com`, which is the domain that will be used for the public API. An export of the notebook for the timeframe of a week is provided in section A.8.

Furthermore, logging has been implemented across all services to provide visibility for system behavior and potential failures. All logs are collected and forwarded to the centralised ELK stack at Sandbox Interactive. This allows for real-time analysis and supports troubleshooting when needed.

8.4 Upcoming Testing

Although various testing and monitoring mechanisms have already been implemented, further measures will be necessary to enhance system robustness before the public API is made available to external developers. One of the planned improvements is the introduction of automated API testing using Postman or similar tools. This will allow for systematic validation of API endpoints, ensuring they return correct and expected responses under different conditions. Automated regression tests will also be added to detect unintended changes in behaviour after future code modifications.

In addition to functional testing, security assessments will be performed. Once the API is deployed in production, a penetration test will be conducted to identify vulnerabilities that attackers could exploit, ensuring that appropriate security measures are in place.

Finally, user testing will be carried out in collaboration with selected third-party developers. By providing beta access to external users, valuable feedback can be gathered on usability, performance, and potential edge cases that may not have been accounted for during internal testing. This iterative testing approach will contribute to a more stable and developer-friendly API.

9 Evaluation

This chapter provides an evaluation of the designed and implemented public API for Albion Online. The evaluation assesses the API's functionality, performance, security, and maintainability against the requirements defined in chapter 5. Additionally, a comparison to the current state described in chapter 4 highlights the API's strengths and areas for improvement. Finally, identified limitations are discussed to guide future development efforts.

9.1 Functional Evaluation

The API was designed to fulfill a set of functional requirements, ensuring secure, structured data access for Albion Online players. The testing efforts (see chapter 8) indicate that the following functionalities have been implemented successfully:

- The API provides structured access to kill-related data, supporting filtering, sorting, and pagination as outlined in FR-1 and FR-7.
- Secure authentication was implemented through API keys, addressing FR-2.
- Rate limiting per IP and API key was introduced to prevent abuse and ensure fair usage, fulfilling FR-3.
- Logging and monitoring mechanisms were integrated, ensuring observability and early failure detection as per FR-4.
- The API supports versioning to maintain backward compatibility for future iterations, aligning with FR-5.

Despite these achievements, some functions remain incomplete or are untested at the time of writing. Further testing measures are yet to be fully implemented to reliably verify and validate the pending requirements. Additional security enhancements are planned, and caching will likely be implemented (FR-6).

Furthermore, additional use cases are implied through FR-1, requiring access to marketplace data, static game data and player, guild and alliance statistics via the API. Kill data access was implemented as an MVP for these other datasets, as defined in FR-1-MVP. The system architecture (section 6.1) is designed to handle these use cases as well. Until a project for additional data sets in the Albion Online Public API is started, this claim cannot be evaluated.

9.2 Non-Functional Evaluation

Beyond functional requirements, the API was designed to meet several non-functional requirements related to performance, maintainability, and security. The evaluation of these aspects is presented in the following paragraphs.

Performance

Initial performance tests in the local development environment and on staging indicate that the API meets NFR-1, with response times within acceptable limits. However, large-scale real-world usage has yet to be assessed. In the live environment, measured response times will likely vary due to different CPU and memory resources, different load on the corresponding services, and because queries in larger database tables will take a longer time.

Maintainability

The maintainability of the newly developed system according to ISO Standard 25010 is required by NFR-11. As explained in subsection 4.1.5, maintainability is characterised by a systems modularity, reusability, analysability, modifiability and testability. Several measures have been implemented to support these characteristics.

The system architecture (section 6.1) is designed to provide strong modularity. During the implementation process, it was easy to work on the required components separately. Communication between the components is limited to distinct interfaces, and changing implementations of these interfaces had no impact on the components that depend on them. This assessment is based on development experiences and remains subjective until further data accumulates. Further development in the future will provide more insight. Evaluating the reusability of the Public API system results in the same claims. These two characteristics also support the compliance with NFR-2: "The systems MUST be extensible for more use cases".

As implied by chapter 8, the analysability, modifiability and testability of the systems can and should be improved in the future. While the applied testing, monitoring, and logging measures provide a base level of these characteristics, more automated testing should be implemented, and further informative metrics and logs would provide quicker and simpler insights into arising issues.

Compliance with best practices

Through the application of code reviews and by adhering to the API design defined in section 6.2, the compliance with NFR-5, NFR-7 and NFR-8 is intended to be high, but cannot be proven.

Localisation

The key management interface has already been translated into all languages that Albion Online supports, but the API documentation has not been translated yet. This remains an open task for the project, only fulfilling half of NFR-6.

Data Consistency

The ETL process ensures that API responses align with in-game data and has been proven to prepare data in time to be considered up-to-date (NFR-9). The execution of the test plan described in section 8.1 will support these claims.

While many of the non-functional requirements have been addressed, additional testing and optimisation are needed to ensure and prove full compliance.

9.3 Comparison to the Game Info Service

The newly developed public API for Albion Online is intended to replace and extend the functionality of the Game Info Service (GIS) API, which is currently used by third-party developers to access game-related data. While the GIS provides a working solution, it had several limitations (see section 4.1) that the new API aims to overcome. This section compares both systems in terms of functionality, security, maintainability, scalability and performance.

Functionality

As listed in subsection 4.1.1, the GIS provided access to various data sets, a lot more than is covered by the system implemented in the scope of this thesis. While it is intended to add further use cases to the new API as per FR-1, a lot of work still needs to be done to fully replace the GIS in functionality.

A function that is missing in the GIS API is versioning. This has been implemented in the system already, providing a functional improvement.

Security

One of the most significant improvements over the GIS is the security model. The GIS API had no access control mechanisms, exposing public endpoints without authentication, authorisation or rate limiting. This made it susceptible to abuse and excessive querying. In contrast, the new API enforces authentication through API keys and implements rate limiting per client and IP address to prevent misuse (FR-2, FR-3).

Maintainability

As stated in subsection 4.1.5, the GIS is considered difficult to maintain by ISO Standard 25010. A few improvements have been made over this as described in the previous section, mainly through the introduction of proper logging, monitoring and testing practices. Improvement in modularity and reusability are claimed in the same evaluation but have not been proven.

Additionally, just like GIS, the new system is lacking developer documentation as of now. Documentation is part of the requirements (FR-10) and will be derived from this thesis once it has been handed in.

Another improvement over the GIS is the compliance with best practices and REST principles. As discussed in the previous section, measures were taken to ensure this compliance, but the fulfilment of the corresponding requirements cannot be proven.

Scalability and Performance

The GIS API was not designed with scalability in mind, leading to performance degradation under high query loads. It relied on a monolithic architecture that made horizontal scaling difficult. In contrast, the new API follows a microservices approach, allowing individual components to scale as needed. According to the tests done so far, the new API complies with NFR-1, responding within one second. This is another improvement over the GIS, which has an average response time of 9 seconds in the American data center.

In subsection 4.1.4, it was also noted that a restart of the GIS in America results in a 30-minute downtime. This will not be the case for the new system, as all involved services restart within a few seconds.

Overall, the new API introduces multiple improvements over the GIS, addressing its limitations in security, scalability, and maintainability. While further testing and feature expansion is required, the system provides a strong foundation for a replacement for the GIS.

9.4 Comparison to other APIs

Compared to other official game APIs, such as those provided by EVE Online, Guild Wars 2, and World of Warcraft (see section 2.2), the Albion Online API introduces similar capabilities but is still in its early stages. While initial endpoints focus solely on kill data, competing game APIs offer broader scopes, including market data, player profiles, and inventory information. Future expansions aim to close this gap.

Furthermore, the API implements authentication through API keys and a rate-limiting mechanism, aligning with the practices observed in the examined APIs in section 2.2 and subsection 2.3.1.

9.5 Limitations

While the current state of the project provides a solid foundation for further development and finally a complete Public API for Albion Online, several limitations have been identified:

- Limited Endpoint Scope: The current implementation only includes kill-related data. Expanding to other aspects of the game, such as marketplace transactions and player progression, would increase its utility.
- Caching: No caching has been implemented for the kill data yet, but will be applied in the future, as it is a part of the requirements for the project FR-6.
- Testing Coverage: While unit and integration tests ensure baseline functionality, comprehensive API testing (e.g., automated Postman tests and penetration testing) is not yet complete.
- Security: Although API key authentication is in place, the safety of the applied security measures has not been tested yet. Security is therefore not guaranteed as of now, which should be changed in the future.
- Performance at Scale: Initial performance benchmarks suggest that the system can handle expected loads, but real-world usage patterns might reveal bottlenecks requiring optimisation.
- User Feedback and Community Engagement: The API has not yet been deployed to external users, meaning real-world feedback is still lacking. Engaging third-party developers early could help refine usability and feature priorities.

Addressing these limitations will be crucial for the API's maintainability and adoption. Future work should focus on extending functionality, improving security, and incorporating user feedback.

10 Conclusion

This thesis presented the design, implementation, and evaluation of a scalable and secure public API for Albion Online, addressing the need for structured and reliable access to in-game data. The project was motivated by the limitations of the existing Game Info Service, which lacked flexibility, scalability, and formal documentation. By adopting modern API design principles, implementing an ETL-based data pipeline, and ensuring compliance with security and performance best practices, the new system provides a robust foundation for external developers and community-driven applications.

10.1 Summary of Contributions

The primary contributions of this thesis include:

- A comprehensive analysis of existing public game APIs and the specific needs of Albion Online.
- The design of a modular system architecture that supports scalability, maintainability, and extensibility.
- The implementation of an ETL process that efficiently transforms and prepares game data for external consumption.
- The integration of security measures, including authentication, authorisation, and rate limiting, to ensure controlled access and prevent abuse.
- The evaluation of the API against functional and non-functional requirements, including performance benchmarks, security considerations, and maintainability analysis.

The developed API successfully meets its primary objectives by providing a scalable, secure and maintainable data access mechanism. It enables third-party developers to create tools that enhance the player experience while maintaining the integrity and security of game data.

10.2 Limitations and Challenges

Despite its successes, the project faced several limitations. Due to time constraints, certain features remain incomplete or untested. Automated API testing, comprehensive security assessments, and a more detailed setup of the API documentation have yet to be implemented. Additionally, while the system was designed with scalability in mind, real-world usage patterns and load testing in a production environment will reveal potential bottlenecks.

Another challenge was data processing. A timely solution to prepare all required data was achieved, but the implementation process required multiple iterations of performance improvements, which took up a lot of time at a critical point in this project.

10.3 Future Work

Several avenues for future improvement remain:

- **Feature Expansion:** Additional endpoints should be introduced to provide access to marketplace transactions, player profiles, and other relevant game data.
- **Comprehensive Testing:** Further unit, integration, API-level and security tests should be implemented to ensure reliability across all endpoints.
- **Performance Optimisation:** Real-world performance monitoring should be conducted to identify potential scaling issues and optimise query efficiency.
- **Community Feedback:** Engaging third-party developers and gathering feedback will be crucial in refining and prioritizing future improvements.

By iterating on these aspects, the Albion Online Public API can evolve into a comprehensive and reliable platform that serves the needs of both the game's community and the development team. This thesis provides a strong foundation for future work, demonstrating the feasibility and benefits of structured data access in massively multiplayer online games.

10.4 Final Remarks

The development of a public API is not merely a technical endeavour but also a strategic decision that shapes how a game interacts with its player base and third-party developers. By providing structured data access and maintaining control over security and performance, this project paves the way for a more open and dynamic ecosystem around Albion Online. With continued improvements and community involvement, the API has the potential to become an indispensable tool for players, developers, and the game's long-term evolution.

Sources

[1] ArenaNet. *API:Main - Guild Wars 2 Wiki (GW2W)*. 2024. URL: https://wiki.guildwars2.com/wiki/API:Main.

[2] baeldung. *A Comparison Between Spring and Spring Boot*. 2024. URL: https://www.baeldung.com/spring-vs-spring-boot.

[3] baeldung. *Batch Insert/Update with Hibernate/JPA*. 2024. URL: https://www.baeldung.com/jpa-hibernate-batch-insert-update.

[4] baeldung. *Learn JPA & Hibernate Series*. 2023. URL: https://www.baeldung.com/learn-jpa-hibernate.

[5] Wayne Barker. *The Hidden Monolith*. 2018. URL: http://www.waynecliffordbarker.co.za/2018/08/01/the-hidden-monolith/.

[6] Richard A. Bartle. *Players who suit MUDs*. 1999. URL: https://mud.co.uk/richard/hcds.htm.

[7] Berk. *Spring Framework vs Dropwizard*. 2023. URL: https://medium.com/@berkdotai/spring-framework-vs-dropwizard-1977469284a4.

[8] Tim Berners-Lee, Roy Thomas Fielding, and Larry Melvin Masinter. *RFC 3986: Uniform resource identifier (URI): Generic syntax*. 2005. URL: https://datatracker.ietf.org/doc/html/rfc3986.

[9] Blizzard Entertainment. *World of Warcraft - Documentation*. 2024. URL: https://develop.battle.net/documentation/world-of-warcraft.

[10] Joshua Bloch. "A Brief, Opinionated History of the API". In: QCon New York. 2018. URL: https://www.infoq.com/presentations/history-api/.

[11] Broadcom. *Spring Batch Introduction :: Spring Batch*. 2025. URL: https://docs.spring.io/spring-batch/reference/spring-batch-intro.html.

[12] Broadcom. *Spring Cloud Gateway*. 2025. URL: https://spring.io/projects/spring-cloud-gateway.

[13] Broadcom. *Spring.io*. 2017. URL: https://spring.io/.

[14] Cambridge University Press. *MMORPG*. 2013. URL: https://dictionary.cambridge.org/dictionary/english/mmorpg.

[15] Cambridge University Press. *Sandbox*. 2013. URL: https://dictionary.cambridge.org/dictionary/english/sandbox.

[16] CCP Games. *EVE API Guide - EVE University Wiki*. 2021. URL: https://wiki.eveuniversity.org/EVE_API_Guide.

[17] Vivian Chen et al. "Enjoyment or Engagement? Role of Social Interaction in Playing Massively Mulitplayer online Role-Playing Games (MMORPGS)". In: *Proceedings of the 5th international conference on Entertainment Computing*. Vol. 4161. Jan. 2006, pp. 262–267. ISBN: 978-3-540-45259-1. DOI: 10.1007/11872320_31.

[18] Ed D. Hardt. *RFC 6749 - The OAuth 2.0 Authorization Framework*. 2012. URL: https://datatracker.ietf.org/doc/html/rfc6749.

[19] Datadog. *Infrastructure & Application Monitoring as a Service | Datadog*. 2025. URL: https://www.datadoghq.com/product/.

[20] Mykyta Divieiev and Viktorija Grigaliūnaitė. "The Impact of Videogame Elements in MMORPG on Consumer Retention". In: *Management of Organizations: Systematic Research* 91 (Dec. 2024), pp. 23–36. DOI: 10.2478/mosr-2024-0002.

[21] Adam DuVander. *API Keys vs OAuth Tokens vs JSON Web Tokens*. 2017. URL: https://zapier.com/engineering/apikey-oauth-jwt/.

[22] Elasticsearch. *What is Elasticsearch?* 2025. URL: https://www.elastic.co/guide/en/elasticsearch/reference/current/elasticsearch-intro-what-is-es.html.

[23] Elasticsearch B.V. *Logstash: Collect, Parse, Transform Logs | Elastic*. 2025. URL: https://www.elastic.co/logstash.

[24] Ethan02, Engelbert Tristram. *T4A - API info*. 2024. URL: https://www.tools4albion.com/api_info.php.

[25] Roy Thomas Fielding. "Architectural Styles and the Design of Network-based Software Architectures". PhD thesis. University of California, Irvine, 2000. URL: https://www.ics.uci.edu/~fielding/pubs/dissertation/top.htm.

[26] Roy Thomas Fielding. *REST APIs must be hypertext-driven*. 2008. URL: https://roy.gbiv.com/untangled/2008/rest-apis-must-be-hypertext-driven.

[27] GitHub, Inc. *GitHub REST API documentation*. 2025. URL: https://docs.github.com/en/rest.

[28] Google Cloud. *What Is A Relational Database (RDBMS)?* URL: https://cloud.google.com/learn/what-is-a-relational-database.

[29] Google For Developers. *Google Maps Platform Documentation*. URL: https://developers.google.com/maps/documentation.

[30] J. Gough, D. Bryant, and M. Auburn. *Mastering API Architecture: Design, Operate, and Evolve API-Based Systems*. O'Reilly Media, 2021. ISBN: 9781492090588.

[31] Grinding Gear Games. *Developer Docs - Path of Exile*. 2025. URL: https://www.pathofexile.com/developer/docs.

[32] K. Hunter. *Irresistible APIs: Designing web APIs that developers will love*. Manning, 2016. ISBN: 9781638353447.

[33] IBM. *What is a relational database?* 2021. URL: https://www.ibm.com/think/topics/relational-databases.

[34] insomnia.rest. *The Collaborative API Development Platform - Insomnia*. 2025. URL: https://insomnia.rest/.

[35] *ISO/IEC 25010:2023, Systems and software engineering — Systems and software Quality Requirements and Evaluation (SQuaRE) — Product quality model*. 2023.

[36] Gita Jackson. *What is API design?* 2025. URL: https://www.ibm.com/think/topics/api-design.

[37] Jenkins. *Jenkins*. 2025. URL: https://www.jenkins.io/.

[38] Brenda Jin, Saurabh Sahni, and Amir Shevat. *Designing Web APIs: Building APIs that developers love*. O'Reilly Media, 2018.

[39] Simon Kemp. *Digital 2024 April Global Statshot Report*. Datareportal, 2024. URL: https://datareportal.com/reports/digital-2024-april-global-statshot.

[40] N. Madden. *API Security in Action*. Manning, 2020. ISBN: 9781617296024.

[41] M. Masse. *REST API Design Rulebook: Designing Consistent RESTful Web Service Interfaces*. O'Reilly Media, 2011. ISBN: 9781449319908.

[42] Nat Sakimura Michael B. Jones John Bradley. *RFC 7519 - JSON Web Token (JWT)*. 2015. URL: https://datatracker.ietf.org/doc/html/rfc7519.

[43] MMO Populations. *Server Population & Player Count - MMO Populations*. 2025. URL: https://mmo-population.com/list.

[44] Badr Nasslahsen. *OpenAPI 3 Library for spring-boot*. 2025. URL: https://springdoc.org/.

[45] Jakob Nielsen. *Usability Engineering*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 1994. ISBN: 9780080520292.

[46] Oracle. *What Is a Relational Database? (RDBMS)?* 2021. URL: https://www.oracle.com/database/what-is-a-relational-database/.

[47] OWASP API Security Project team. *OWASP API Security Top 10*. 2023. URL: https://owasp.org/API-Security/editions/2023/en/0x00-header/.

[48] OWASP Top 10 team. *OWASP Top 10:2021*. 2021. URL: https://owasp.org/Top10/.

[49] M. H. Kassab P. A. Laplante. *Requirements Engineering for Software and Systems*. Fourth Edition. Auerbach Publications, 2022. ISBN: 9781032275994.

[50] Matt Palmer. *Understanding ETL*. O'Reilly Media, Inc, 2024.

[51] PayPal. *Get Started with PayPal REST APIs*. URL: https://developer.paypal.com/api/rest/.

[52] postman.com. *Postman: The World's Leading API Platform*. 2025. URL: https://www.postman.com/.

[53] Puppetlabs. *GitHub - puppetlabs/puppet: Server automation framework and application*. 2024. URL: https://github.com/puppetlabs/puppet.

[54] Redis. *Document Database - Redis*. 2021. URL: https://redis.io/nosql/document-databases/.

[55] J. Reis and M. Housley. *Fundamentals of Data Engineering*. O'Reilly Media, 2022. ISBN: 9781098108274.

[56] L. Richardson, M. Amundsen, and S. Ruby. *RESTful Web APIs: Services for a Changing World*. O'Reilly Media, 2013. ISBN: 9781449359744.

[57] Leonard Richardson. "Justice Will Take Us Millions Of Intricate Moves". In: QCon. 2008. URL: https://www.crummy.com/writing/speaking/2008-QCon/act3.html.

[58] Carlos Rodriguez et al. "REST APIs: A Large-Scale Analysis of Compliance with Principles and Best Practices". In: June 2016, pp. 21–39. ISBN: 978-3-319-38790-1. DOI: 10.1007/978-3-319-38791-8_2.

[59] Peter Rottmann. *apiDoc - Inline Documentation for RESTful web APIs*. 2015. URL: https://apidocjs.com/.

[60] David Salz. *Secrets of Albion Online, Part 1*. 2016. URL: https://davidsalz.de/secrets-of-albion-online-part-1/.

[61] David Salz and Robin Henkys. "Making an Independent MMO - The Albion Online Story". In: Devcom developer conference. 2018. URL: https://www.slideshare.net/slideshow/making-an-independend-mmo-the-albion-online-story/127399111.

[62] Sandbox Interactive GmbH. *Albion East is Open!* 2023. URL: https://albiononline.com/news/albion-east-live.

[63] Sandbox Interactive GmbH. *Albion Europe is Open!* 2024. URL: https://albiononline.com/news/albion-europe-live.

[64] Sandbox Interactive GmbH. *Albion Online - The Journey Starts NOW!* 2017. URL: https://albiononline.com/news/albion-online-official-release.

[65] Sandbox Interactive GmbH. *Albion Online Breaks All-Time Player Record — Again!* 2024. URL: https://albiononline.com/news/record-player-numbers.

[66] Sandbox Interactive GmbH. *Albion Online is Now Free-to-Play*. 2019. URL: https://albiononline.com/news/albion-online-is-now-free-to-play.

[67] Sandbox Interactive GmbH. *Albion Online's Mobile Version is Out Now!* 2021. URL: https://albiononline.com/news/mobile-version-out-now.

[68] Smilegate RPG. *Lostark OpenAPI Developer Portal*. 2025. URL: https://developer-lostark.game.onstove.com/getting-started.

[69] Kai Spichale. *API-Design : Praxishandbuch für Java- und Webservice-Entwickler / Kai Spichale*. 2., überarbeitete und erweiterte Auflage. Heidelberg: dpunkt-Verlag, 2019. ISBN: 9783864906114.

[70] Phil Sturgeon. *Build APIs you won't hate: Everyone and their dog wants an API, so you should probably learn how to build them*. Lean Publishing, 2015.

[71] A.S. Tanenbaum and M. van Steen. *Distributed Systems: Principles and Paradigms*. Createspace Independent Publishing Platform, 2016. ISBN: 9781530281756.

[72] The Albion Online Data Project. *Home | The Albion Online Data Project*. 2025. URL: https://www.albion-online-data.com/.

[73] The MapStruct authors. *MapStruct – Java bean mappings, the easy way!* 2025. URL: https://mapstruct.org/.

[74] The PostgreSQL Global Development Group. *PostgreSQL: About*. 2025. URL: https://www.postgresql.org/about/.

[75] The PostgreSQL Global Development Group. *PostgreSQL: The world's most advanced open source database*. 2025. URL: https://www.postgresql.org/about/.

[76] S. Tilkov, M. Eigenbrodt, and S. Schreier. *REST und HTTP: Entwicklung und Integration nach dem Architekturstil des Web*. dpunkt-Verlag, 2015. ISBN: 9783864901201.

[77] Timbatao. *Is it allowed to scan your internet trafic and pick up logs*. 2017. URL: https://forum.albiononline.com/index.php/Thread/51604-Is-it-allowed-to-scan-your-internet-trafic-and-pick-up-logs/?pageNo=1.

[78] Twilio Inc. *Docs: API Reference, Tutorials, and Integration | Twilio*. 2025. URL: https://www.twilio.com/docs.

[79] Twilio Inc. *SendGrid v3 API Documentation*. 2025. URL: https://www.twilio.com/docs/sendgrid/api-reference.

[80] Valve Corporation. *Steam Web API Documentation*. URL: https://steamcommunity.com/dev.

[81] J. Webber, S. Parastatidis, and I. Robinson. *REST in Practice: Hypermedia and Systems Architecture*. Theory in practice series. O'Reilly Media, 2010. ISBN: 9781449396923.

[82] Kevin Wittek. *Spring JPA – Multiple Databases*. 2024. URL: https://www.baeldung.com/spring-data-jpa-multiple-databases.

[83] X Corp. *Twitter API Documentation*. 2025. URL: https://developer.x.com/en/docs/x-api.

[84] Zalando SE Opensource. *Zalando RESTful API and Event Guidelines*. URL: https://opensource.zalando.com/restful-api-guidelines.

Glossary

Disclaimer: Glossary entries were written by ChatGPT and only slightly altered by the author.

Alliance A collection of multiple guilds working together under a shared banner. Alliances enable large-scale cooperation in PvP conflicts, particularly in territorial warfare and endgame content. 37, 38, 41, 49, 90

Arena A controlled PvP environment where players compete in structured matches without the full risks of open-world combat. Albion Online's arena mode allows players to test builds, practice combat skills, and earn rewards in a setting where they do not lose all their equipment on death. 37

Base64 An encoding scheme used to convert binary data into an ASCII string format. It is commonly used for encoding data in URLs, email attachments, and data transmission over text-based protocols. 20

Battle A broader engagement involving multiple players or groups fighting in an area. Battles in Albion Online range from small skirmishes to large-scale fights between guilds and alliances over territory or resources. 37, 38, 41

Beta A phase of game development where the game is made available to a limited group of players for testing before its official release. 89

Botting The practice of using automated scripts or programs (bots) to perform repetitive in-game actions, such as farming resources, trading, or engaging in combat. Botting is often used to gain unfair advantages and is typically against a game's terms of service. 1, 10, 12

Build (or Character Build) The combination of gear, abilities, and playstyle a player uses in combat. In Albion Online, builds are highly customizable, as abilities are tied to equipment rather than character classes, allowing for flexible roles in PvE and PvP encounters. 37–39, 41, 45

Eavesdropping A form of cyber attack where an unauthorized third party intercepts and listens to communication between two parties, often to steal sensitive data such as credentials, personal information, or encryption keys. This can occur on unencrypted networks or through compromised communication channels. 21

Fame A form of experience in Albion Online, earned through various activities such as combat, gathering, crafting, and trading. Fame is required to progress in the Destiny Board and to unlock stronger gear and abilities. 37, 38, 41, 63

Full-Loot A game mechanic in Albion Online where defeated players drop all equipped and inventory items upon death. This system increases the risk and reward of PvP engagements, encouraging strategic gameplay and resource management. 4

Guild A player-formed organization that provides benefits such as shared resources, coordinated activities, and structured PvP engagements. Guilds play a crucial role in Albion Online's economy and warfare, competing for territory and influence. 5, 15, 37, 38, 41, 49, 67, 90

Kill The act of defeating another player in combat, resulting in their character's death. 37–39, 41, 42, 59–63, 67, 69, 76–82, 85, 87

Man-in-the-Middle (MitM) A security attack where an attacker secretly intercepts and possibly alters the communication between two parties without their knowledge. This can allow the attacker to eavesdrop, manipulate transmitted data, or impersonate one of the parties to gain unauthorized access to sensitive information. 21

Mockup A static or interactive visual representation of a system, interface, or component that is used to demonstrate design, layout, or functionality without implementing actual logic or backend functionality. 64

Network Scraping A technique used to extract data by intercepting and analysing network traffic between a client and a server. In online games, network scraping is often employed to gain unauthorized access to game data, which can lead to security vulnerabilities and unfair advantages. 1, 10

PvE (Player versus Environment) A mode of gameplay in which players engage with computer-controlled enemies. 37

Replication A database strategy where data is copied and maintained across multiple servers to enhance availability, reliability, and fault tolerance. Replication can be synchronous, ensuring immediate consistency, or asynchronous, allowing for eventual consistency across distributed systems. 27, 55

Sharding A database partitioning technique used to distribute large datasets across multiple databases or servers. Sharding improves scalability and performance by reducing the load on individual database nodes, with each shard handling a subset of the overall data. 27

Single Responsibility Principle A software design principle that states that a class or module should have only one reason to change, meaning it should have only one responsibility. 45

Twitch A live streaming platform primarily focused on video game content, including esports competitions, game development streams, and community-driven broadcasts. Twitch enables interactive engagement through live chat and monetization options for streamers. 8, 41

A Appendix

A.1 Sample Kill Response from the Game Info Service

```
{
  "groupMemberCount": 2,
  "numberOfParticipants": 3,
  "EventId": 123456789,
  "TimeStamp": "2023-01-01T12:00:00.000000000Z",
  "Version": 4,
  "Killer": {
    "AverageItemPower": 950.0,
    "Equipment": {
      "MainHand": {
        // item data
      },
      "OffHand": {
        // item data
      },
      "Head": {
        // item data
      },
      "Armor": {
        // item data
      },
      "Shoes": {
        // item data
      },
      "Bag": {
        // item data
      },
      "Cape": {
        // item data
       },
      "Mount": {
        // item data
      },
      "Potion": {
        // item data
       },
```

```
      "Food": {
        // item data
      }
    },
    "Inventory": [],
    "Name": "Player123",
    "Id": "UUID-12345",
    "GuildName": "GuildA",
    "GuildId": "UUID-GUILD-1",
    "AllianceName": "AllianceX",
    "AllianceId": "UUID-ALLIANCE-1",
    "Avatar": "AVATAR_DEFAULT",
    "AvatarRing": "AVATARRING_STANDARD",
    "DeathFame": 0,
    "KillFame": 50000,
    "FameRatio": 500000.00
  },
  "Victim": {
    "AverageItemPower": 900.0,
    "Equipment": {
      "MainHand": {
        // item data
      },
      "OffHand": null,
      "Head": {
        // item data
      },
      "Armor": {
        // item data
      },
      "Shoes": {
        // item data
      },
      "Bag": {
        // item data
      },
      "Cape": {
        // item data
      },
      "Mount": {
        // item data
      },
      "Potion": null,
      "Food": {
        // item data
      }
    },
    "Inventory": [
      {
```

```
        // item data
      },
      {
        // item data
      }
    ],
    "Name": "Player456",
    "Id": "UUID-67890",
    "GuildName": "GuildB",
    "GuildId": "UUID-GUILD-2",
    "AllianceName": "AllianceY",
    "AllianceId": "UUID-ALLIANCE-2",
    "Avatar": "AVATAR_WARRIOR",
    "AvatarRing": "AVATARRING_EVENT",
    "DeathFame": 50000,
    "KillFame": 0,
    "FameRatio": 0.00
  },
  "TotalVictimKillFame": 50000,
  "Location": null,
  "Participants": [
    {
      "AverageItemPower": 940.0,
      "Equipment": {
        "MainHand": {
          // item data
      },
        "OffHand": {
          // item data
        }
      },
      "Name": "Player789",
      "Id": "UUID-11223",
      "GuildName": "GuildA",
      "GuildId": "UUID-GUILD-1",
      "AllianceName": "AllianceX",
      "AllianceId": "UUID-ALLIANCE-1",
      "Avatar": "AVATAR_KNIGHT",
      "AvatarRing": "AVATARRING_RARE",
      "DamageDone": 7500.0,
      "SupportHealingDone": 1200.0
    }
  ],
  "GroupMembers": [
    {
      "Name": "Player123",
      "Id": "UUID-12345",
      "GuildName": "GuildA",
      "GuildId": "UUID-GUILD-1",
```

```
      "AllianceName": "AllianceX",
      "AllianceId": "UUID-ALLIANCE-1",
      "KillFame": 50000
    }
  ],
  "BattleId": 987654321,
  "KillArea": "OPEN_WORLD",
  "Category": "PVP",
  "Type": "KILL"
}
```

Resources/code_snippets/gis_kill.json

A.2 SecurityConfig class in the API Gateway

```
package com.albiononline.apigateway.config;

// imports removed for brevity

@Setter
@Getter
@Configuration
@ConfigurationProperties(prefix = "albion.api-gateway.security")
@EnableWebFluxSecurity
@EnableReactiveMethodSecurity
public class SecurityConfig {

    private static final String OAUTH_CSTOOL_SCOPE_PREFIX = "cstool_";

    private final ApiKeyConverter apiKeyConverter;

    private URI csToolHost;
    private List<UserConfig> users;

    public SecurityConfig(ApiKeyConverter apiKeyConverter) {
        this.apiKeyConverter = apiKeyConverter;
    }

    @Bean
    public static PasswordEncoder passwordEncoder() {
        return PasswordEncoderFactories.createDelegatingPasswordEncoder();
    }

    @Bean
    public SecurityWebFilterChain springSecurityFilterChain(
    ServerHttpSecurity http, AuthServiceClient authServiceClient) {

        if (csToolHost != null) {
            UriComponents uri = UriComponentsBuilder.fromUri(csToolHost)
                    .path("/.well-known/jwks.json")
```

```
                .build();

        http
                .oauth2ResourceServer()
                .jwt()
                .jwkSetUri(uri.toUriString())
                .jwtAuthenticationConverter(jwtAuthenticationConverter())
;
    }

    http.csrf().disable();
    http.formLogin().disable();
    http.logout().disable();
    http.httpBasic();

    AuthenticationWebFilter apiKeyAuthFilter = new
AuthenticationWebFilter(new KeyAuthenticationManager(authServiceClient));
    apiKeyAuthFilter.setServerAuthenticationConverter(apiKeyConverter);

    http.securityMatcher(ServerWebExchangeMatchers.pathMatchers("/v*/
admin/**", "/actuator/refresh"))
            .authorizeExchange()
            .pathMatchers("/actuator/refresh").hasRole(Roles.
RELOAD_CONFIG.getName())
            .pathMatchers("/v*/admin/**").hasRole(Roles.ADMIN.getName());

    http.securityMatcher(ServerWebExchangeMatchers.pathMatchers( "/**"))
            .authorizeExchange()
            .pathMatchers("/v*/*/forZendesk").permitAll()
            .pathMatchers("/v*/admin/**").permitAll()
            .pathMatchers(HttpMethod.OPTIONS).permitAll()
            .pathMatchers("/swagger","/swagger/**", "/v3/api-docs/**").
permitAll()
            .pathMatchers("/*/swagger","/*/swagger/**").permitAll()
            .pathMatchers("/actuator/info").permitAll()
            .anyExchange().authenticated()
            .and()
            .addFilterAt(apiKeyAuthFilter, SecurityWebFiltersOrder.
AUTHENTICATION);


    return http.build();
}

@Bean
@ConditionalOnProperty(prefix = "albion.api-gateway.security", name = "
users[0].username")
public MapReactiveUserDetailsService userDetailsService() {
    List<UserDetails> userDetails = users.stream()
```

```
                .map(user -> User
                        .withUsername(user.getUsername())
                        .password(user.password)
                        .roles(user.roles)
                        .build()
                )
                .collect(Collectors.toList());

        return new MapReactiveUserDetailsService(userDetails);
    }

    @Bean
    protected Converter<Jwt, Mono<AbstractAuthenticationToken>>
    jwtAuthenticationConverter() {
        ReactiveJwtAuthenticationConverter jwtAuthenticationConverter = new
    ReactiveJwtAuthenticationConverter();

        jwtAuthenticationConverter.setJwtGrantedAuthoritiesConverter(
    jwtGrantedAuthoritiesConverter());

        return jwtAuthenticationConverter;
    }

    @Bean
    protected Converter<Jwt, Flux<GrantedAuthority>>
    jwtGrantedAuthoritiesConverter() {
        JwtGrantedAuthoritiesConverter grantedAuthoritiesConverter = new
    JwtGrantedAuthoritiesConverter();
        grantedAuthoritiesConverter.setAuthorityPrefix(
    OAUTH_CSTOOL_SCOPE_PREFIX);

        return new ReactiveJwtGrantedAuthoritiesConverterAdapter(
    grantedAuthoritiesConverter);
    }

    @Getter
    @Setter
    public static class UserConfig {

        private String username;
        private String password;
        private String[] roles;

        public void setPassword(String password) {
            this.password = passwordEncoder().encode(password);
        }
    }
}
```

Resources/code_snippets/SecurityConfig.java.txt

A.3 IpAndApiKeyResolver class in the API Gateway

```java
package com.albiononline.apigateway.ratelimit;

// imports removed for brevity

@Component
public class IpAndApiKeyResolver implements KeyResolver {

    public static final String API_KEY_HEADER = "x-albion-api-key";

    @Override
    public Mono<String> resolve(ServerWebExchange exchange) {
        ServerHttpRequest request = exchange.getRequest();

        String remoteIp = getRemoteIp(request);
        String apiKey = getApiKey(request);

        String path = request.getURI().getPath();
        if (remoteIp.isEmpty()
            || (apiKey.isEmpty() && !StringUtils.containsIgnoreCase(path, "swagger-ui"))) {
            return Mono.empty();
        }

        return Mono.just(remoteIp + apiKey);
    }

    private String getRemoteIp(ServerHttpRequest request) {
        if (request != null
            && request.getRemoteAddress() != null
            && request.getRemoteAddress().getAddress() != null
            && StringUtils.isNotBlank(request.getRemoteAddress().getAddress().getHostAddress())) {

            return request.getRemoteAddress().getAddress().getHostAddress();
        }
        return "";
    }

    private String getApiKey(ServerHttpRequest request) {
        List<String> apiKey = request.getHeaders().get(API_KEY_HEADER);
        if (!CollectionUtils.isEmpty(apiKey) && StringUtils.isNotBlank(apiKey.get(0))) {
            return apiKey.get(0);
        }
        return "";
    }
```

```
}
```

Listing A.1: *IpAndApiKeyResolver.java; Mostly written by Jakob Erdmann and Ivan Borgardt*

A.4 Database Schema for Kills

```
CREATE TABLE locations
(
    id                   BIGSERIAL PRIMARY KEY,
    cluster_display_name TEXT,
    cluster_name         TEXT,
    type                 TEXT,
    continent            TEXT,
    cluster_type         TEXT,
    x_coordinate         DOUBLE PRECISION,
    y_coordinate         DOUBLE PRECISION
);

CREATE TABLE players
(
    id                 BIGSERIAL PRIMARY KEY,
    name               TEXT,
    character_id       UUID,
    guild_name         TEXT,
    guild_id           UUID,
    alliance_name      TEXT,
    alliance_id        UUID,
    avatar             TEXT,
    avatar_ring        TEXT,
    average_item_power DOUBLE PRECISION
);

CREATE TABLE items
(
    id             BIGSERIAL PRIMARY KEY,
    item_type      TEXT,
    count          INTEGER,
    quality        INTEGER,
    legendary_soul JSONB,
    active_spells  JSONB,
    passive_spells JSONB
);

CREATE INDEX items_item_type_index
    ON items (item_type);

CREATE TABLE player_equipment
(
    id             BIGSERIAL PRIMARY KEY,
    player         BIGINT NOT NULL
```

```
        CONSTRAINT player_equipment_players_id_fk
            REFERENCES players,
    item           BIGINT NOT NULL
        CONSTRAINT player_equipment_items_id_fk
            REFERENCES items,
    equipment_slot TEXT
);

CREATE INDEX player_equipment_player_index
    ON player_equipment (player);

CREATE TABLE player_inventory
(
    id     BIGSERIAL PRIMARY KEY,
    player BIGINT NOT NULL
        CONSTRAINT player_inventory_players_id_fk
            REFERENCES players,
    item   BIGINT NOT NULL
        CONSTRAINT player_inventory_items_id_fk
            REFERENCES items
);

CREATE INDEX player_inventory_player_index
    ON player_inventory (player);

CREATE TABLE kills
(
    id                     BIGSERIAL PRIMARY KEY,
    event_id               BIGINT,
    kill_timestamp         TIMESTAMP,
    total_victim_kill_fame BIGINT,
    kill_area              TEXT,
    location               BIGINT
        CONSTRAINT kills_locations_id_fk
            REFERENCES locations,
    killer                 BIGINT
        CONSTRAINT kills_killer_fk
            REFERENCES players,
    victim                 BIGINT
        CONSTRAINT kills_victim_fk
            REFERENCES players
);

CREATE INDEX kills_kill_timestamp_index
    ON kills (kill_timestamp);

CREATE INDEX kills_total_victim_kill_fame_index
    ON kills (total_victim_kill_fame);

```

```
CREATE INDEX kills_event_id_index
    ON kills (event_id);

CREATE TABLE kill_participants
(
    id                   BIGSERIAL PRIMARY KEY,
    kill                 BIGINT NOT NULL
        CONSTRAINT kill_participants_kills_id_fk
            REFERENCES kills,
    player               BIGINT NOT NULL
        CONSTRAINT kill_participants_players_id_fk
            REFERENCES players,
    damage_done          DOUBLE PRECISION,
    support_healing_done DOUBLE PRECISION
);

CREATE INDEX kill_participants_kill_index
    ON kill_participants (kill);

CREATE TABLE kill_group_members
(
    id     BIGSERIAL PRIMARY KEY,
    kill   BIGINT NOT NULL
        CONSTRAINT kill_participants_kills_id_fk
            REFERENCES kills,
    player BIGINT NOT NULL
        CONSTRAINT kill_group_members_players_id_fk
            REFERENCES players
);

CREATE INDEX kill_group_members_kill_index
    ON kill_group_members (kill);
```

Resources/code_snippets/db_schema.sql.txt

A.5 Kill Mapper

```
package com.albiononline.statsetlservice.domain.mapper;

// imports removed for brevity

@Mapper(unmappedTargetPolicy = ReportingPolicy.IGNORE,
    nullValuePropertyMappingStrategy = NullValuePropertyMappingStrategy.
    IGNORE)
public interface KillMapper {

    KillMapper INSTANCE = Mappers.getMapper(KillMapper.class);

    @Mapping(target = "killTimestamp", source = "timeStamp")
    @Mapping(target = "killer.characterId", source = "killer.id")
```

```
    @Mapping(target = "victim.characterId", source = "victim.id")
    @Mapping(target = "location.XCoordinate", source = "location.x")
    @Mapping(target = "location.YCoordinate", source = "location.y")
    @Mapping(target = "location.clusterType", source = "location.cType")
    Kill fromEventData(KillEventData killEventData);

    @Mapping(target = "itemType", source = "type")
    Item fromKillEventItem(KillEventItem killEventItem);

    default List<PlayerEquipment> fromKillEventItemMap(
        Map<String, KillEventItem> killEventItemMap
    ) {
        if (killEventItemMap == null) {
            return new ArrayList<>();
        }
        List<PlayerEquipment> playerEquipments = new ArrayList<>();
        for (Map.Entry<String, KillEventItem> entry : killEventItemMap.
    entrySet()) {
            if (entry.getValue() != null) {
                PlayerEquipment playerEquipment = new PlayerEquipment();
                playerEquipment.setItem(fromKillEventItem(entry.getValue()));
                playerEquipment.setEquipmentSlot(entry.getKey());
                playerEquipments.add(playerEquipment);
            }
        }
        return playerEquipments;
    }

    default List<PlayerInventory> fromInventoryArray(
        KillEventItem[] killEventItemArray
    ) {
        List<PlayerInventory> playerInventoryList = new ArrayList<>();
        for (KillEventItem killEventItem : killEventItemArray) {
            if (killEventItem != null) {
                PlayerInventory playerInventory = new PlayerInventory();
                playerInventory.setItem(fromKillEventItem(killEventItem));
                playerInventoryList.add(playerInventory);
            }
        }
        return playerInventoryList;
    }

    @Mapping(target = "characterId", source = "id")
    Player playerFromEventGroupMember(
        KillEventGroupMember killEventGroupMember
    );

    default KillGroupMember fromEventGroupMember(
        KillEventGroupMember killEventGroupMember
```

```
    ) {
        KillGroupMember killGroupMember = new KillGroupMember();
        Player player = playerFromEventGroupMember(killEventGroupMember);
        killGroupMember.setPlayer(player);
        return killGroupMember;
    }

    @Mapping(target = "characterId", source = "id")
    Player playerFromEventParticipant(
        KillEventParticipant killEventParticipant
    );

    default KillParticipant fromEventParticipant(
        KillEventParticipant killEventParticipant
    ) {
        if (killEventParticipant == null) {
            return null;
        }

        KillParticipant killParticipant = new KillParticipant();

        killParticipant.setDamageDone(killEventParticipant.getDamageDone());
        killParticipant.setSupportHealingDone(
            killEventParticipant.getSupportHealingDone()
        );
        killParticipant.setPlayer(
            playerFromEventParticipant(killEventParticipant)
        );

        return killParticipant;
    }
}
```

Listing A.2: *KillMapper.java*

A.6 Spring Batch configuration for kill event processing

```
package com.albiononline.statsetlservice.config;

// imports removed for brevity

@Configuration
@EnableBatchProcessing
@RequiredArgsConstructor
@ConfigurationProperties(prefix = "albion.stats-etl-service.kill-processing")
public class KillEventBatchConfiguration {

    private final KillEventProcessor killEventProcessor;
    private final KillEventRepository killEventRepository;
    private final KillRepository killRepository;
```

```
14      private final JobRepository jobRepository;
15      private final PlatformTransactionManager statsTransactionManager;
16
17      @Getter
18      @Setter
19      private int readerPageSize = 1000;
20      @Getter
21      @Setter
22      private int stepChunkSize = 100;
23
24      @Bean
25      @StepScope
26      public RepositoryItemReader<KillEvent> killEventReader(@Value("#{
        jobParameters['index']}") long index) {
27          return new RepositoryItemReaderBuilder<KillEvent>()
28                  .repository(killEventRepository)
29                  .name("killEventReader")
30                  .methodName("findByIdGreaterThan")
31                  .arguments(index)
32                  .pageSize(readerPageSize)
33                  .sorts(Collections.singletonMap("id", Sort.Direction.ASC))
34                  .build();
35      }
36
37      @Bean
38      public RepositoryItemWriter<Kill> killWriter() {
39          return new RepositoryItemWriterBuilder<Kill>()
40                  .repository(killRepository)
41                  .build();
42      }
43
44      @Bean
45      public Step killEventBatchStep(ItemReader<KillEvent> killEventReader,
        ItemWriter<Kill> killWriter) {
46          return new StepBuilder("killEventBatchStep")
47                  .<KillEvent, Kill>chunk(stepChunkSize)
48                  .reader(killEventReader)
49                  .processor(killEventProcessor)
50                  .writer(killWriter)
51                  .repository(jobRepository)
52                  .transactionManager(statsTransactionManager)
53                  .build();
54      }
55
56      @Bean
57      public Job killEventBatchJob(Step killEventBatchStep,
        ProcessShutdownListener processShutdownListener) {
58          return new JobBuilder("killEventBatchJob")
59                  .repository(jobRepository)
```

```
                .listener(processShutdownListener)
                .start(killEventBatchStep)
                .incrementer(new RunIdIncrementer())
                .build();
    }

    @Bean
    public JobRegistryBeanPostProcessor jobRegistryBeanPostProcessor(
    JobRegistry jobRegistry) {
        JobRegistryBeanPostProcessor postProcessor = new
    JobRegistryBeanPostProcessor();
        postProcessor.setJobRegistry(jobRegistry);
        return postProcessor;
    }

}
```

Listing A.3: *KillEventBatchConfiguration.java*

A.7 Kill Processing Batch Scheduler

```
package com.albiononline.statsetlservice;

// imports removed for brevity

@Slf4j
@Component
@RequiredArgsConstructor
@ConfigurationProperties(prefix = "albion.stats-etl-service.kill-processing")
public class KillBatchScheduler {

    private final KillEventRepository killEventRepository;
    private final KillRepository killRepository;
    private final KillMappingService killMappingService;
    private final ExecutorService taskExecutor;

    @Getter
    @Setter
    private int readerPageSize = 100;

    private Long lastProcessedEventId = null;

    @Scheduled(fixedDelayString = "${albion.stats-etl-service.kill-processing
    .delay-in-ms:5000}")
    public void processKillBatch() {
        if (lastProcessedEventId == null) {
            lastProcessedEventId = killRepository.findTopByOrderByEventIdDesc
    ()
                    .map(Kill::getEventId)
                    .orElse(0L);
```

```
28          }
29
30          Slice<KillEvent> killEvents = killEventRepository.findByIdGreaterThan
    (
31                  lastProcessedEventId, PageRequest.of(0, readerPageSize, Sort.
    Direction.ASC, "id")
32          );
33
34          if (killEvents.isEmpty()) {
35              return;
36          }
37
38          CompletableFuture.runAsync(() -> killMappingService.
    processKillEventsBatch(killEvents.getContent()), taskExecutor)
39                  .exceptionally(ex -> {
40                      log.error("ERROR_PROCESSING_KILL_BATCH", ex);
41                      return null;
42                  });
43
44          lastProcessedEventId = killEvents.getContent().get(killEvents.
    getContent().size() - 1).getId();
45      }
46  }
```

Listing A.4: *KillBatchScheduler.java*

A.8 Public API Metrics Notebook in DataDog

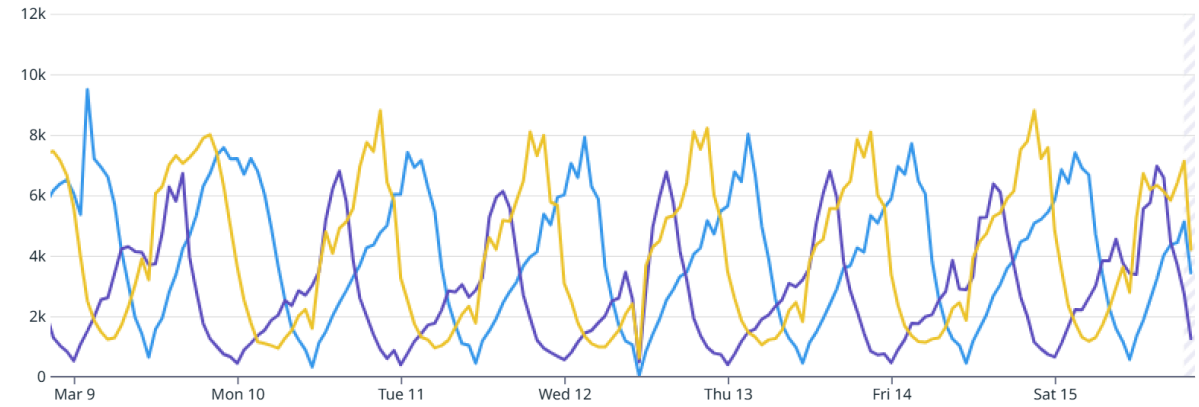
1w Mar 8, 8:35 pm – Mar 15, 8:35 pm

Open Notebook in Datadog

Public API Metrics

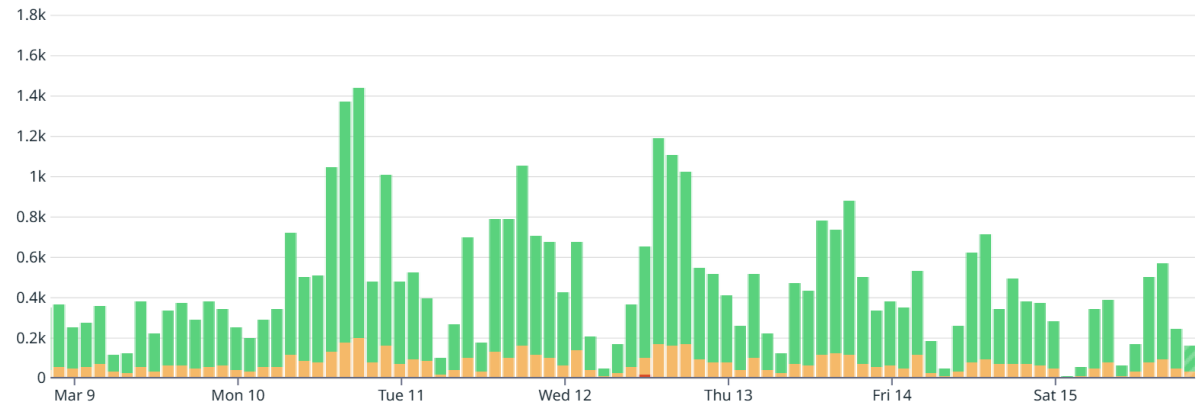
Created by **Pauline Röhr** on Feb 20

Kill Processing



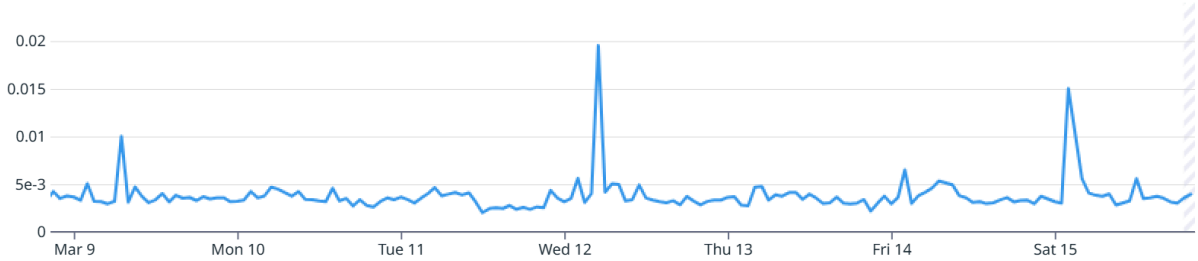
TAGS	METRIC	AVG	MIN	MAX	SUM	VALUE
*	US	4.02k	28	9.49k	676k	7.40k
*	Singapore	2.77k	390	6.96k	465k	2.21k
*	Amsterdam	4.06k	619	8.80k	682k	1.77k

Requests api.albiononline.com



TAGS	METRIC	AVG	MIN	MAX	SUM	VALUE
*	2xx	387.7	6	1,237	32.57k	47.0
*	4xx	70.5	3	199	5.93k	7.0
*	5xx	15.0	15	15	15	—

Response time of api.albiononline.com



A.9 Full Source Code

The source code for this project has been uploaded to the Sandbox Interactive Google Drive as a password protected zip file, but is not available publicly.

A.10 Tools Used

For the work on this thesis, the following tools were used:

- **Jira and Confluence:** For project management
- **Miro:** To create diagrams
- **ChatGPT and Grammarly:** For wording, sentence construction and grammatical/spelling correction
- **JetBrains IntelliJ:** The main IDE for this project
- **JetBrains Rider:** Utilised to go through game server code
- **Jetbrains DataGrip:** Database management (mostly PostgreSQL)
- **Overleaf:** As LaTeX editor

Other tools not mentioned here are sufficiently introduced in the thesis.

Statutory Declaration

I herewith formally declare that I have written the submitted thesis independently. I did not use any outside support except for the quoted literature and other sources mentioned in the paper. I clearly marked and separately listed all of the literature and other sources I employed when producing this academic work, either literally or in content. I am aware that the violation of this regulation will lead to failure of the thesis.

_____

City, Date

_____

Signature

Confidentiality Notice

This master thesis contains confidential and proprietary information of Sandbox Interactive GmbH.

This work may only be made available to the primary and secondary reviewers and authorised members of the board of examiners. Any publication and duplication of this master thesis – even in part – is prohibited.

An inspection of this work by third parties requires the expressed permission of the author and involved company.

_____

City, Date

_____

Signature